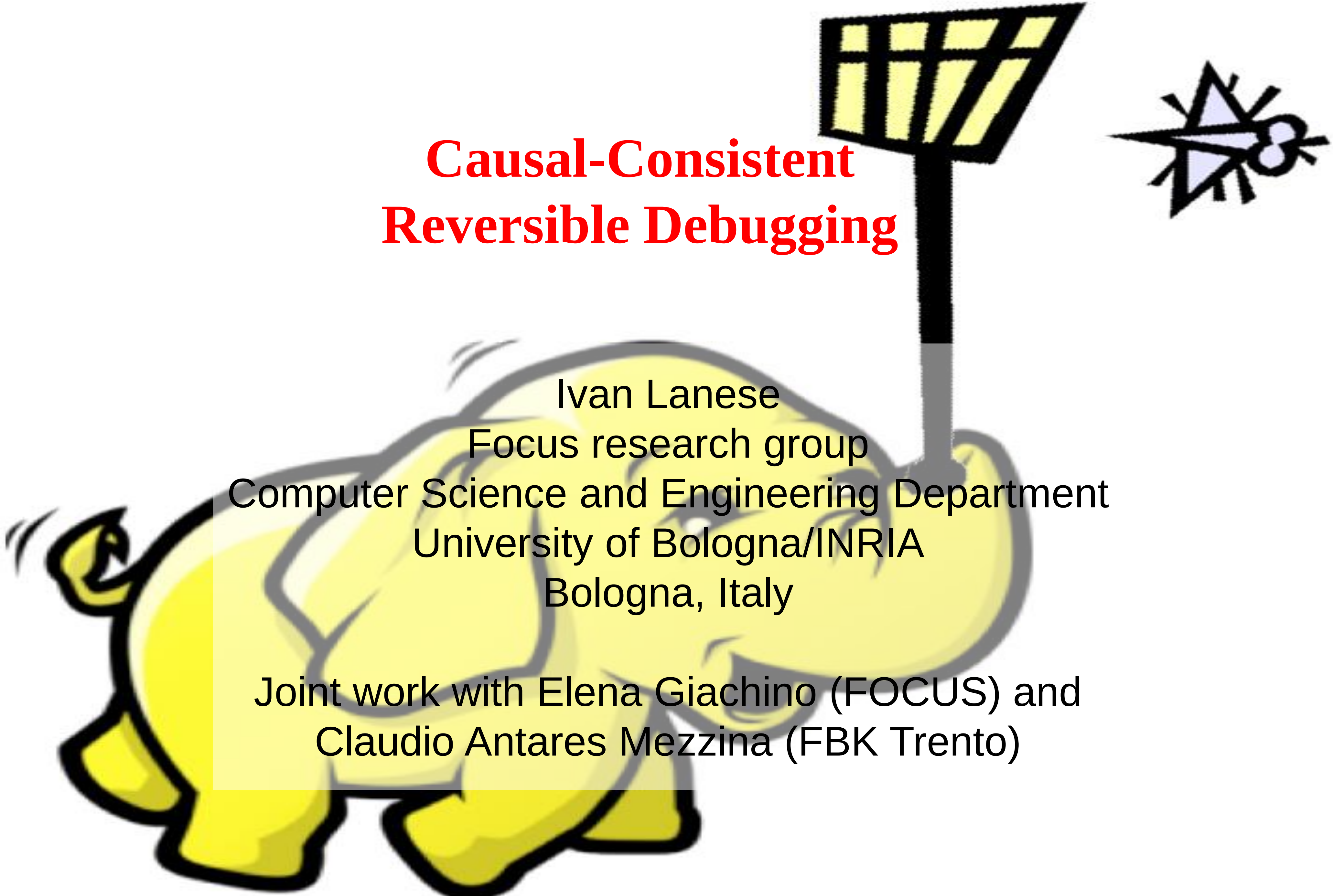
Causal-Consistent Reversible Debugging



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

Joint work with Elena Giachino (FOCUS) and
Claudio Antares Mezzina (FBK Trento)

Reversibility for debugging

- Our claim:
  Causal-consistent reversibility can help the programmer to debug concurrent applications
- Reversibility helpful also in a sequential setting
  - Some commercial debuggers provide the command "step back" in a sequential setting
  - For instance, gdb
- Can we apply similar techniques in a concurrent setting?
- Can we do better?

Debugging and causality

- Debugging amounts to find the bug that caused a given misbehavior
- Debugging strategy: follow causality links backward from misbehavior to bug
- Which primitives do we need?
- Mainly, primitives that given a misbehavior go back towards its causes
- We define them in the context of 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
- Thread-based concurrency
- Asynchronous communication via ports
- Shared memory
  - Variable names are sent, not their content
- Variables are always created fresh and never modified
- Higher-order language
  - Procedures can be communicated

Main primitives

- One primitive for each possible misbehavior
- Wrong value in a variable: **rollvariable id** goes to the state just before the creation of variable **id**
- Wrong value in a queue element: **rollsend id n** undoes the last **n** sends to port **id**
  - If **n** is unspecied, the last send is undone
- Thread blocked on a receive: **rollreceive id n** undoes the last **n** reads on the port **id**
  - If **n** is unspecied, the last receive is undone
- Unexpected thread: **rollthread t** undoes the creation of thread **t**

Using causal-consistent primitives

- The programmer can follow causality links backward
- No need for the programmer to know which thread or instruction originated the misbehavior
  - The primitives find them
- The procedure can be iterated till the bug is found
- Only relevant actions are undone
  - All the primitives in the last slide are causal consistent

Additional commands

- A debugger needs two kinds of commands
  - Commands to control execution
  - Commands to explore the configuration
    - Both code and state
- Some of the commands are standard, others related to reversibility or causal consistency

Execution control commands: standard

- Step forward
  - The user specifies the target thread
- Run
  - Round robin scheduler
- Breakpoints

Execution control commands: non standard

- Step backward
  - Goes back one step
  - 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
  - Only in reversible debuggers
- **Roll t n**
  - Undoes the last **n** actions of thread **t**
  - Causal consistent
  - Only in causal-consistent reversible debuggers

Configuration exploration commands

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

Testing our approach

- According to [Lu et al., ASPLOS 08] most of the concurrency bugs
  - Involve only 2 threads, and a few variables
  - Are order violation, atomicity violation or deadlock
- The causal-consitent approach allows the programmer to concentrate on the involved threads
- We put our debugger at work on three paradigmatic examples, one for each class of common bugs

CaReDeb: a causal-consistent debugger

- Only a prototype to test our ideas
- Debugs μOz programs
- Written in Java
- Based on the reversible semantics of μOz
- Available at
  `http://proton.inrialpes.fr/~mezzina/deb/`
- Starts with `java -jar deb.jar inputfile`

CaReDeb: a causal-consistent debugger

DEMO

Future work

- Improving the prototype
  - Usability: integration with eclipse
  - Performance: both in time and in space
- Extending the language
  - Additional constructs
  - What about applying the approach to a mainstream language?
- Assessment
  - Can we improve the rate of bug discovery thanks to our approach?

Finally

Thanks!

Questions?

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
$$\frac{\langle \textbf{skip}\ T\rangle}{0} \,\Big\|\, \frac{T}{0}$$

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

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

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

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

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

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

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

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

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	$\dfrac{t[H\ \mathbf{skip}]C}{0}$	$\dfrac{t[H]\langle \mathbf{skip}\ C\rangle}{0}$
R:bk:var	$\dfrac{t[H\ \ast x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{x = v}$	$\dfrac{t[H]\langle \mathbf{let}\ x = v\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0}$
R:bk:npr	$\dfrac{t[H\ \ast x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \xi \parallel \xi : c}$	$\dfrac{t[H]\langle \mathbf{let}\ x = c\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0}$
R:bk:npt	$\dfrac{t[H\ \ast x]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \xi \parallel \xi : \bot \vert \bot}$	$\dfrac{t[H]\langle \mathbf{let}\ x = \mathtt{NewPort}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{0}$
R:bk:if1	$\dfrac{t[H\ \mathbf{if}(x)S_2]\langle S_1\ \langle \mathbf{esc}\ C\rangle\rangle}{x = \mathbf{true}}$	$\dfrac{t[H]\langle \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ C\rangle}{x = \mathbf{true}}$
R:bk:nth	$\dfrac{t[H\ \ast t']C \parallel t'[\bot]\langle S\ \langle\rangle\rangle}{0}$	$\dfrac{t[H]\langle \mathbf{thread}\ S\ \mathbf{end}\ C\rangle}{0}$
R:bk:pc	$\dfrac{t[H\ \{\ x\ (x_i)_1^n\ \}]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{0}$	$\dfrac{t[H]\langle \{\ x\ (x_i)_1^n\ \}\ C\rangle}{0}$
R:bk:snd	$\dfrac{t[H\ \uparrow x]C}{x = \xi \parallel \xi : t\!:\!y; K \vert K_h}$	$\dfrac{t[H]\langle \{\ \mathtt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K \vert K_h}$
R:bk:rcv	$\dfrac{t[H\ \downarrow x(z)]\langle S\ \langle \mathbf{esc}\ C\rangle\rangle}{z = w \parallel x = \xi \parallel \xi : K \vert t'\!:\!y, t; K_h}$	$\dfrac{t[H]\langle \mathbf{let}\ z = \{\ \mathtt{Receive}\ x\ \}\ \mathbf{in}\ S\ \mathbf{end}\ C\rangle}{x = \xi \parallel \xi : K; t'\!:\!y \vert K_h}$
R:bk:scp	$\dfrac{t[H\ \mathbf{esc}]C}{0}$	$\dfrac{t[H]\langle \mathbf{esc}\ C\rangle}{0}$