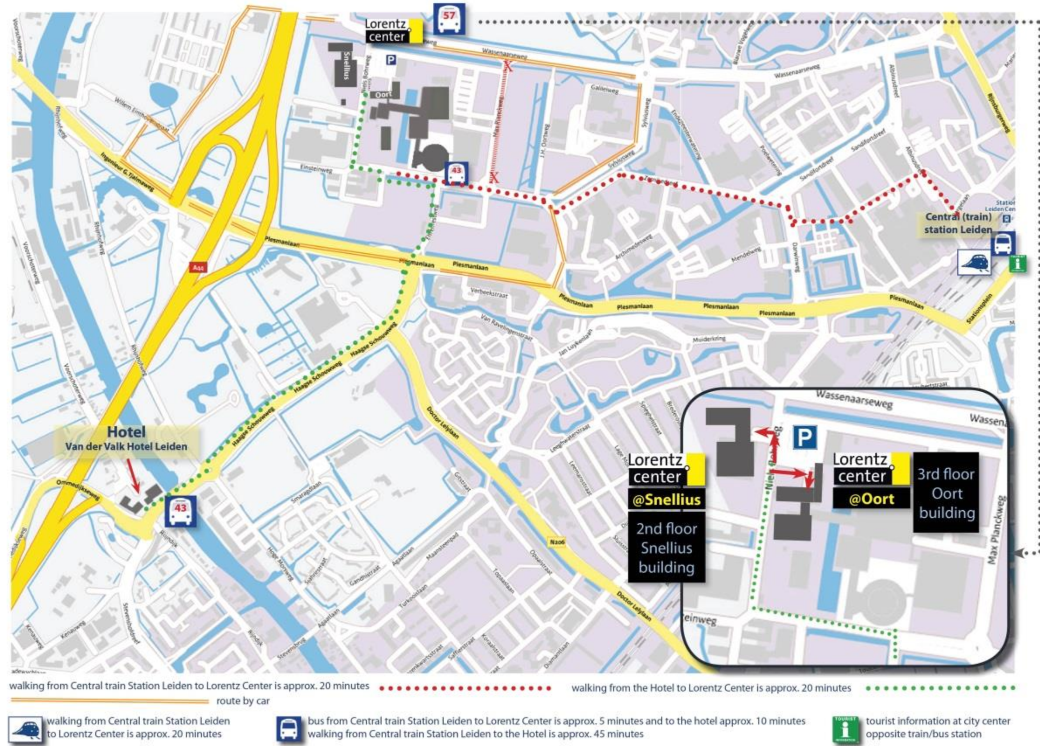
# Reversible Debugging

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

Joint work with Elena Giachino
(Univ. Bologna/INRIA, Italy) and Claudio
Antares Mezzina (IMT Lucca, Italy)

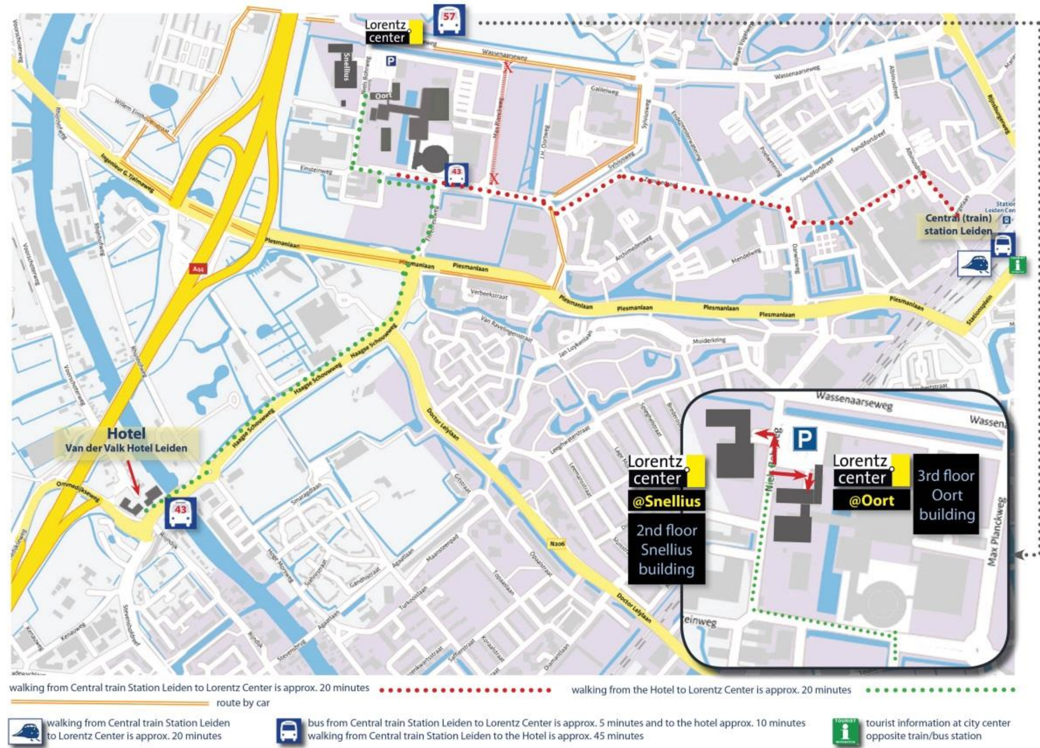
# Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach:
  causal-consistent reversible debugging
- Conclusion



# Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach:
  causal-consistent reversible debugging
- Conclusion



# Why debugging?

- Developers spend 50% of their programming time finding and fixing bugs
- The global cost of debugging has been estimated in $312 billions annually
- The cost of debugging is bound to increase with the increasing complexity of software
  - Size
  - Concurrency, distribution
- Surprisingly, a very little amount of research concentrates on debugging

# Standard debugging strategy

- When a failure occurs, one has to re-execute the program with a breakpoint before the expected bug
- Then one executes step-by-step forward from the breakpoint, till the bug is found
- Limitations:
  - High cost of replaying
    - » Time, use of the actual execution environment
  - Difficult to precisely replay the execution
    - » Concurrency or non-determinism
  - Difficult to find the exact point where to put the breakpoint
    - » If breakpoint too late, the execution needs to be redone
    - » Frequently, many attempts are needed

# (Sequential) Reversibility

- Reversibility allows one to execute programs not only in the standard, forward, direction, but also backward, going to past states
- Reversibility in a sequential setting: recursively undo the last action
- Many actions lose information: one needs to keep information on the past states
  - The assignment x=0 deletes the old value of x, which should be saved
- Reversibility has applications in hardware design, biological modeling, simulation, quantum computing...

# Reversibility for debugging

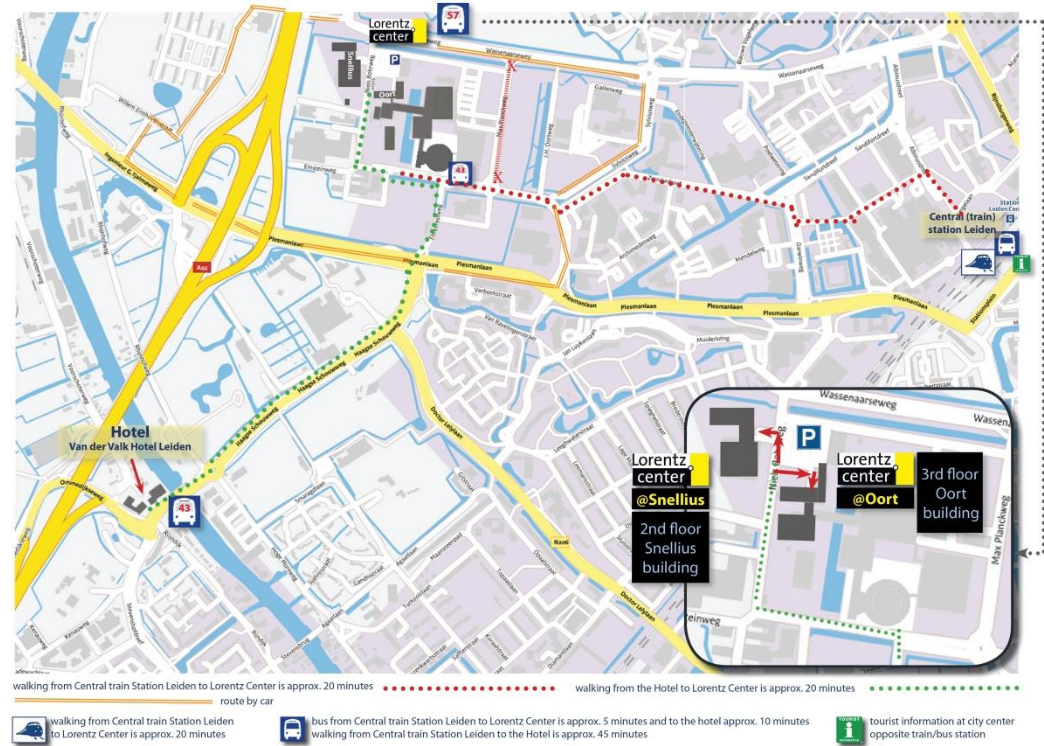
- Reversible debuggers extend standard debuggers with the ability to execute the program under analysis also backward
- Avoids the common “Damn, I put the breakpoint too late” error
  - Just execute backward from where the program stopped till the desired point is reached
  - You may even not need a breakpoint: if the program crashed, you can execute backward from where it crashed
- The overhead due to storing history information is a main limitation for reversible debuggers

# Reversible debugging strategy

- Take an execution containing a failure and move backward and forward along it looking for the bug
- The exact same execution can be explored many times forward and backward
  - Non-determinism is no more a problem
- Some form of causal exploration is also possible
  - If variable $i$ has not the desired value, run backward with a watchpoint telling where the value of $i$ changes

# Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach:
  causal-consistent reversible debugging
- Conclusion



# A reversible debugger: GDB

- GDB supports reversible debugging since version 7.0 (2009)
- Uses record and replay
  - One activates the recording modality
  - Executes the program forward
  - Can explore the recorded execution backward and forward
  - When exploring, instructions are not re-executed

# GDB reverse commands

- Like the forward commands (step, next, continue), but in the backward direction
- Reverse-step: goes back to the last instruction
- Reverse-next: goes back to the last instruction, does not go inside functions
- Reverse-continue: runs back till the last stop event
- ...
- Breakpoints and watchpoints can be used also in backward direction

# A commercial reversible debugger: UndoDB

- From UndoSoftware, Cambridge, UK
  http://undo-software.com/
- Built as an extension of GDB
- Available for Linux and Android
- Allows reversible debugging for programs in C/C++

# UndoDB commands

- Recording enabled by default, but can be deferred
- Reversible commands from GDB (reimplemented)
- Commands for exploring the recorded execution, more high-level
  - Define and go to bookmark
  - Go back n "simulated nanoseconds"
- Various commands for configuration
- Commands to write a recorded execution to file, and reload it
  - Useful to record on client premises and explore at company premises

# UndoDB winning feature

## Performance

- Comparison with GDB, on recording gzipping a 16MB file

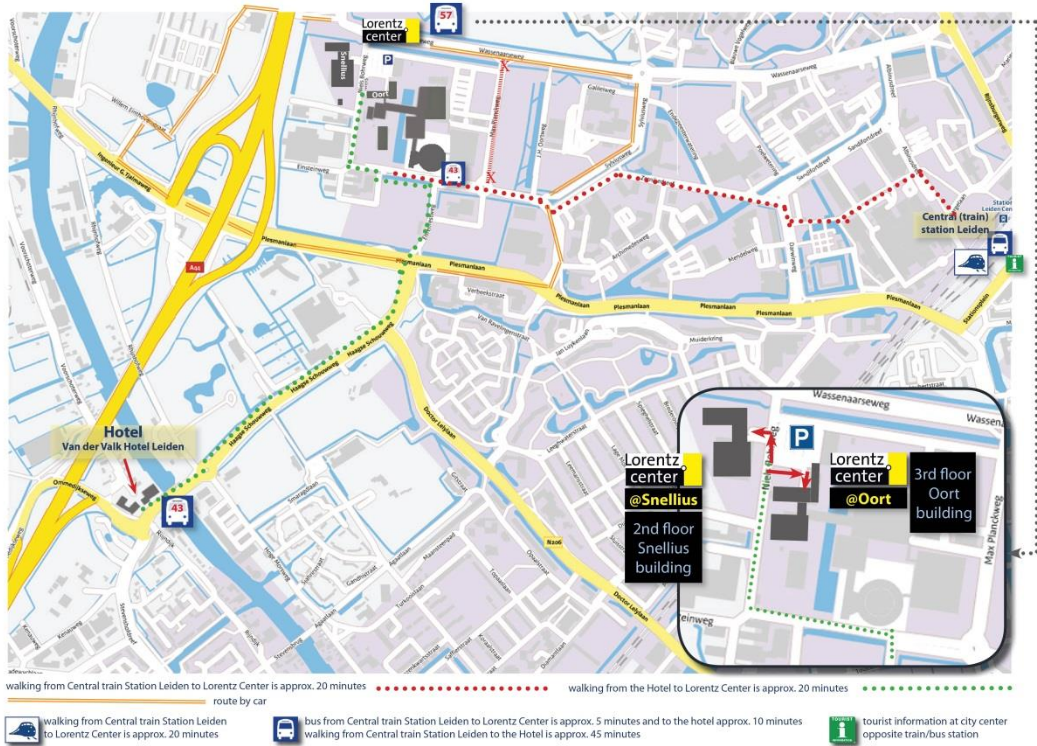
| | Native | UndoDB | GDB |
|---|---|---|---|
| Time | 1.49 s | 2.16 s (1.75 x) | 21 h (50000 x) |
| Space | - | 17.8 MB | 63 GB |

# UndoDB: how it works

- Takes periodically snapshots of memory
  - Only changed pages are stored (copy-on-write)
  - The frequence of snapshots changes dynamically
- Nondeterministic events are stored during recording and simulated during replay
  - Including thread switches, shared memory accesses and system calls
  - The same thread scheduling is considered during replay

# Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach:
  causal-consistent reversible debugging
- Conclusion



# Causal-consistent reversible debugging

- We are interested in debugging concurrent/distributed programs
- The definition "recursively undo the last action" is no more valid: there are many possible last actions
- Since [Danos&Krivine, CONCUR 2004] the notion of reversibility for concurrent systems is causal-consistent reversibility
  - Any action can be undone, provided that its consequences (if any) have been undone
  - Concurrent actions can be undone in any order, but causal-dependent actions are undone in reverse order

# Debugging and causality

- Causal-consistency relates backward computations with causality
- Debugging amounts to find the bug that caused a given misbehavior
- We propose the following debugging strategy: follow causality links backward from misbehavior to bug
- Which primitives do we need to enable such a strategy?
  - The details depend on the chosen language...
  - ...but the general idea carries over to different languages

# Our target language: 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

# The **roll** primitive

- The main primitive we propose is **roll t n**
- Undoes the last **n** actions of thread **t**...
- ... in a causal-consistent way
  - Before undoing an action one has to undo all (and only) the actions depending on it
- A single **roll** may cause undoing actions in many threads

# Different interfaces for **roll**

- One interface 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**
- Thread blocked on a receive: **rollreceive id n** undoes the last **n** reads on the port **id**
- Unexpected thread: **rollthread t** undoes the creation of thread **t**

# Using roll-like 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 automatically
- The procedure can be iterated till the bug is found
- Only relevant actions are undone
  - Thanks to causal consistency
- Looking at which threads are involved gives useful information
  - If an unexpected thread is involved, an interference between the two threads has happened

# Additional commands

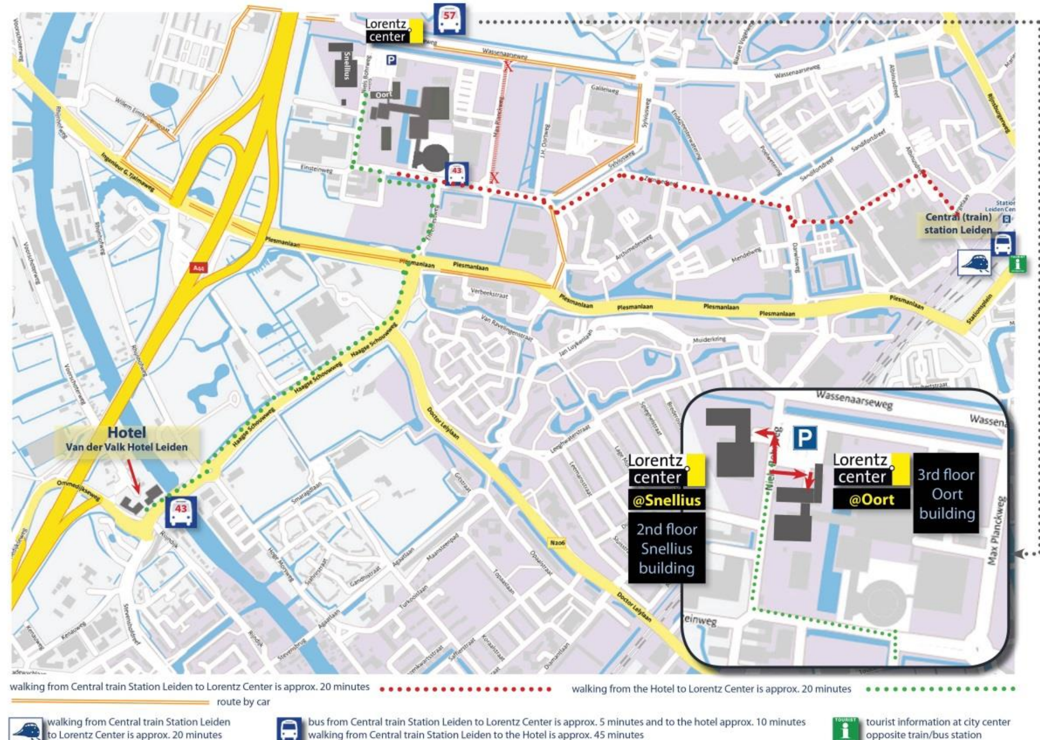
- The roll-like commands are added on top of a standard debugger
  - Commands for forward execution, breakpoints, state exploration
- A few other commands are related to reversibility
  - Step backward
  - History exploration

# CaReDeb: a causal-consistent debugger

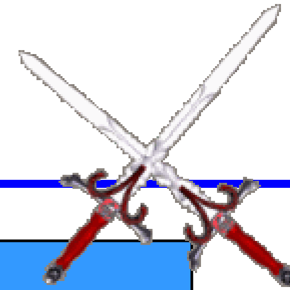
- Only a prototype to test our ideas
- Debugs μOz programs
- Written in Java
- Available at `http://www.cs.unibo.it/caredeb`
- Description and underlying theory in
  [Giachino, Lanese & Mezzina, FASE 2014]

# Roadmap

- Reversible debugging
- Reversible debugging in practice
- An academic approach:
  causal-consistent reversible debugging
- Conclusion



# Research vs industry



| Research | Industry |
|---|---|
| Innovative, complex techniques | Simpler, more standard techniques |
| Toy/limited languages | Real languages |
| No integration in the software development environment | Emphasis on integration in the software development environment |
| Performance not a key | Performance is the key |
| No interest in support and maintenance | Many companies live on support and maintenance |

# Future work

- Extending the set of language constructs we are able to tackle, towards real languages
- Find simplified mechanisms easier to implement efficiently
- Interact with UndoSoftware to bridge the gap
  - We want them to use our techniques, not our debugger

# Some advertising

- We have COST Action IC1405 on Reversible computation – extending horizons of computing
- Not only debugging: reversible languages, reversible automata, reversible circuits, …
- Just preliminary results on verification for reversible systems
- Feel free to contact Irek Ulidowski or me if you want to join

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

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

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

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

# 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 $$\frac{t[H]\langle \mathbf{skip}\ C\rangle}{0} \Bigg\| \frac{t[H\ \mathbf{skip}]C}{0}$$

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

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

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

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

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

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

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

$$\text{if } y' \text{ fresh} \ \wedge\ \theta \triangleq x = \xi \parallel z = w$$

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

# μOz reversible semantics: backward rules

R:bk:skp

$$\frac{t[H\ \mathbf{skip}]C}{0} \;\Big\|\; \frac{t[H]\langle \mathbf{skip}\ C\rangle}{0}$$

R:bk:var

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

R:bk:npr

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

R:bk:npt

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

R:bk:if1

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

R:bk:nth

$$\frac{t[H\ \ *t']C \parallel t'[\bot]\langle S\ \langle\rangle\rangle}{0} \;\Big\|\; \frac{t[H]\langle \mathbf{thread}\ S\ \mathbf{end}\ C\rangle}{0}$$

R:bk:pc

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

R:bk:snd

$$\frac{t[H\ \ \uparrow x]C}{x = \xi \parallel \xi : t{:}y; K|K_h} \;\Big\|\; \frac{t[H]\langle \{\ \mathtt{Send}\ x\ y\ \}\ C\rangle}{x = \xi \parallel \xi : K|K_h}$$

R:bk:rcv

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

R:bk:scp

$$\frac{t[H\ \mathbf{esc}]C}{0} \;\Big\|\; \frac{t[H]\langle \mathbf{esc}\ C\rangle}{0}$$