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 \(\gamma\)
- 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

- $[[P, Q]_\gamma] = \nu a \nu c a\langle 0\rangle^c\langle 0\rangle^0 | a(X) \triangleright_\gamma [P] | c(Y) \triangleright [Q]$
- 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**
  
  \[ P ::= \overline{a} | a.P | P|Q | \nu a P | 0 |[P \triangleright_k Q] | co k \]

- **Semantics**
  
  \[
  \overline{a} | a.P \rightarrow P \\
  [P \triangleright_k Q] | R \rightarrow [P | R \triangleright_k Q | R] \text{ if } k \notin fn(R) \\
  [P| co k \triangleright_k Q] \rightarrow P \\
  [P \triangleright_k Q] \rightarrow Q
  \]

- **Processes from the environment moved into the transaction to interact with it**
  - Saved also in the compensation

- **Implicit abort, explicit commit**
Example: transactions interacting

- $[\bar{a} \triangleright_k Q] | [a. P \triangleright_h Q'] \rightarrow$
  
- $[a. P | [\bar{a} \triangleright_k Q] \triangleright_h Q' | [\bar{a} \triangleright_k Q]] \rightarrow$
  
- $[[\bar{a} | a. P \triangleright_k Q | a. P] \triangleright_h Q' | [\bar{a} \triangleright_k Q ]] \rightarrow$
  
- $[[P \triangleright_k Q | a. P] \triangleright_h Q' | [\bar{a} \triangleright_k Q ]]$

- 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

- $\bar{a} | a.R | [P \triangleright_k Q] \rightarrow$
  
  $[P | \bar{a} | a.R \triangleright_k Q | \bar{a} | a.R] \rightarrow$

  $[P | R \triangleright_k Q | \bar{a} | a.R] \rightarrow$

  $Q | \bar{a} | a.R$

- Why undoing the synchronization on $a$?
- No reason for it to occur inside the transaction
Interacting transactions in croll-$\pi$

- $[[P \triangleright_l Q]] = [vl \llbracket P \rrbracket \mid l(roll \gamma) \mid l(X) \triangleright X, \llbracket Q \rrbracket]_{\gamma}$
- We simulate the automatic abort with a $roll$ that can be enabled at any moment
- $[co l] = l(X) \triangleright 0$
- A commit disables the abort
Comparing the two approaches

- $[[P \triangleright_l Q]] = [\nu l [P] | l\langle\text{roll } \gamma \rangle | l(X) \triangleright X, [Q]]_\gamma$
- In \texttt{croll-}\pi 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 \texttt{croll-}\pi abort is not atomic
  - First, commit becomes impossible
  - Then, abort is performed
- Atomicity problem solvable with choice
  - $\text{roll } \gamma + l(X) \triangleright 0$
  - 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
- 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]
  
  - S₁ S₂ [Sequence]
  
  - let x = v in S end [Variable declaration]
  
  - if x then S₁ else S₂ end [Conditional]
  
  - thread S end [Thread creation]
  
  - let x=c in S end [Procedure declaration]
  
  - \{x x₁ … xₙ\} [Procedure call]
  
  - let x=Newport in S end [Port creation]
  
  - \{Send x y\} [Send]
  
  - let x =\{Receive y\} in S end [Receive]

- c ::= proc \{x₁ … xₙ\} 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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R:skp</strong></td>
<td>$\langle \text{skip } T \rangle \parallel T$</td>
<td>0 0</td>
</tr>
<tr>
<td><strong>R:var</strong></td>
<td>$\langle \text{let } x = v \text{ in } S \text{ end } T \rangle \parallel \langle S^{x'/x} \parallel T \rangle \quad \text{if } x' \text{ fresh}$</td>
<td>$x' = v$</td>
</tr>
<tr>
<td><strong>R:npr</strong></td>
<td>$\langle \text{let } x = c \text{ in } S \text{ end } T \rangle \parallel \langle S^{x'/x} \parallel T \rangle \quad \text{if } x', \xi \text{ fresh}$</td>
<td>$x' = \xi \parallel \xi : c$</td>
</tr>
<tr>
<td><strong>R:npt</strong></td>
<td>$\langle \text{let } x = \text{NewPort} \text{ in } S \text{ end } T \rangle \parallel \langle S^{x'/x} \parallel T \rangle \quad \text{if } x', \xi \text{ fresh}$</td>
<td>$x' = \xi \parallel \xi : \bot$</td>
</tr>
<tr>
<td><strong>R:ifl</strong></td>
<td>$\langle \text{if } x \text{ then } S_1 \text{ else } S_2 \text{ end } T \rangle \parallel \langle S_1 \parallel T \rangle \quad x = \text{true}$</td>
<td>$x = \text{true}$</td>
</tr>
<tr>
<td><strong>R:nth</strong></td>
<td>$\langle \text{thread } S \text{ end } T \rangle \parallel T \parallel \langle S \parallel \rangle$</td>
<td>0 0</td>
</tr>
<tr>
<td><strong>R:pc</strong></td>
<td>$\langle { x \ x_1 \ldots x_n } T \rangle \parallel \langle S^{x_1/y_1} \ldots { x_n/y_n } \parallel T \rangle \quad x = \xi \parallel \xi : \text{proc } { y_1 \ldots y_n } \text{ end}$</td>
<td>$x = \xi \parallel \xi : \text{proc } { y_1 \ldots y_n } \text{ end}$</td>
</tr>
<tr>
<td><strong>R:send</strong></td>
<td>$\langle { \text{Send } x \ y \parallel T \rangle \parallel T \rangle \quad x = \xi \parallel \xi : Q \parallel z = w \quad x = \xi \parallel \xi : y; Q$</td>
<td>$y = \xi \parallel \xi : Q ; z \parallel z = w$</td>
</tr>
<tr>
<td><strong>R:recv</strong></td>
<td>$\langle \text{let } x = { \text{Receive } y } \text{ in } S \text{ end } T \rangle \parallel \langle S^{x'/x} \parallel T \rangle \quad \text{if } x' \text{ fresh}$</td>
<td>$y = \xi \parallel \xi : Q ; z \parallel z = w \parallel x' = w$</td>
</tr>
</tbody>
</table>
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
\[ \frac{t[H]\langle \text{skip } C \rangle}{0} \quad \frac{t[H \text{ skip}]C}{0} \]

R:fw:var
\[ \frac{t[H]\langle \text{let } x = v \text{ in } S \text{ end } C \rangle}{0} \quad \frac{t[H \times x']\langle S\{x'/x\} \langle \text{esc } C \rangle \rangle}{0} \]
if \( x' \) fresh

R:fw:npr
\[ \frac{t[H]\langle \text{let } x = c \text{ in } S \text{ end } C \rangle}{0} \quad \frac{t[H \times x']\langle S\{x'/x\} \langle \text{esc } C \rangle \rangle}{0} \]
if \( x', \xi \) fresh

R:fw:npt
\[ \frac{t[H]\langle \text{let } x = \text{NewPort in } S \text{ end } C \rangle}{0} \quad \frac{t[H \times x']\langle S\{x'/x\} \langle \text{esc } C \rangle \rangle}{0} \]
if \( x', \xi \) fresh

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

R:fw:nth
\[ \frac{t[H]\langle \text{thread } S \text{ end } C \rangle}{0} \quad \frac{t[H \times t']C \parallel t'\langle \bot \rangle\langle S \langle \rangle \rangle}{0} \]
if \( t' \) fresh

R:fw:pc
\[ \frac{t[H]\langle \{ x (x_i)^n \} \ C \rangle}{x = \xi \parallel \xi : \text{proc } \langle (y_i)^n \} S \text{ end} \rangle} \quad \frac{t[H \{ x (x_i)^n \}\langle S\{x/y_i\}^n \langle \text{esc } C \rangle \rangle}{x = \xi \parallel \xi : \text{proc } \langle (y_i)^n \} S \text{ end} \rangle} \]

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

R:fw:rec
\[ \frac{t[H]\langle \text{let } y = \{ \text{Receive } x \} \text{ in } S \text{ end } C \rangle}{\theta \parallel \xi : K; t': z|K_h} \quad \frac{t[H \downarrow x(y')]\langle S\{y'/y\} \langle \text{esc } C \rangle \rangle}{\theta \parallel \xi : K|t' : z,t;K_h \parallel y' = w} \]
if \( y' \) fresh \& \( \theta \triangleq x = \xi \parallel z = w \)

R:fw:scp
\[ \frac{t[H]\langle \text{esc } C \rangle}{0} \quad \frac{t[H \text{ esc}]C}{0} \]
μOz reversible semantics: backward rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Right-hand Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>R:bk:skp</td>
<td>$t[H \ skip]C$</td>
<td>0</td>
</tr>
<tr>
<td>R:bk:var</td>
<td>$t[H \ * x] \langle S \ \langle esc \ C \rangle \rangle$</td>
<td>$t[H] \langle let \ x = v \ in \ S \ end \ C \rangle$</td>
</tr>
<tr>
<td>R:bk:npr</td>
<td>$x = v$</td>
<td>0</td>
</tr>
<tr>
<td>R:bk:npt</td>
<td>$t[H \ * x] \langle S \ \langle esc \ C \rangle \rangle$</td>
<td>$t[H] \langle let \ x = c \ in \ S \ end \ C \rangle$</td>
</tr>
<tr>
<td>R:bk:npt</td>
<td>$x = \xi \</td>
<td></td>
</tr>
<tr>
<td>R:bk:ifl</td>
<td>$t[H \ if(x)S_2] \langle S_1 \ \langle esc \ C \rangle \rangle$</td>
<td>$t[H] \langle if \ x \ then \ S_1 \ else \ S_2 \ end \ C \rangle$</td>
</tr>
<tr>
<td>R:bk:ifl</td>
<td>$x = true$</td>
<td>$x = true$</td>
</tr>
<tr>
<td>R:bk:nth</td>
<td>$t[H \ * t^']C \</td>
<td></td>
</tr>
<tr>
<td>R:bk:pc</td>
<td>$t[H \ { \ x \ (x_i)^n_1 \ } \langle S \ \langle esc \ C \rangle \rangle$</td>
<td>$t[H] \langle { \ x \ (x_i)^n_1 \ } \ C \rangle$</td>
</tr>
<tr>
<td>R:bk:sdn</td>
<td>$t[H \ \uparrow x]C$</td>
<td>$t[H] \langle { \ Send \ x \ y \ } \ C \rangle$</td>
</tr>
<tr>
<td>R:bk:sdn</td>
<td>$x = \xi \</td>
<td></td>
</tr>
<tr>
<td>R:bk:rvn</td>
<td>$t[H \ \downarrow x(z)] \langle S \ \langle esc \ C \rangle \rangle$</td>
<td>$t[H] \langle let \ z = { \ Receive \ x \ } \ in \ S \ end \ C \rangle$</td>
</tr>
<tr>
<td>R:bk:rvn</td>
<td>$z = w \</td>
<td></td>
</tr>
<tr>
<td>R:bk:scp</td>
<td>$t[H \ esc]C$</td>
<td>0</td>
</tr>
</tbody>
</table>
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/](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 mainstream 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?