A reversible debugger for $\mu$Oz

Claudio Antares Mezzina

SOA-FBK

December 10, 2012

joint work with Ivan Lanese
Roadmap

1. μOz

2. Reversible Debuggers

3. μOz reversible debugger
µOz?

- subset of Oz language [Van Roy et al.]

- Higher-Order language
  - thread-based concurrency
  - asynchronous communication via ports (channels)

- µOz advantages:
  - similar to $HO\pi$
  - well-known stack-based abstract machine
\[ S ::= \]

\begin{align*}
& \text{skip} & \text{empty stm} \\
& S_1 \; S_2 & \text{sequence} \\
& \text{let } x = v \text{ in } S \text{ end} & \text{var declaration} \\
& \text{if } x \text{ then } S_1 \text{ else } S_2 \text{ end} & \text{conditional} \\
& \text{thread } S \text{ end} & \text{thread creation} \\
& \text{let } x = c \text{ in } S \text{ end} & \text{procedure declaration} \\
& \{ x \; \tilde{y} \} & \text{procedure call} \\
& \text{let } x = \text{NewPort} \text{ in } S \text{ end} & \text{port creation} \\
& \{ \text{Send } x \; y \} & \text{send on a port} \\
& \text{let } x = \{ \text{Receive } y \} \text{ in } S \text{ end} & \text{receive from a port} \\
\end{align*}

\[ v ::= \text{true} | \text{false} \]

\[ c ::= \text{proc } \{ \tilde{x} \} \; S \; \text{end} \]
programs written as stacks of instructions
a rule transforms a pair \((\text{program}, \text{state})\) into a new pair
variables are \textit{always} created fresh and never \textit{modified}
sent values are variables names, not their contents
\[ \text{R:var} \quad \langle \text{let } x = v \text{ in } S \text{ end } T \rangle \quad \langle S \{ x' / x \} T \rangle \quad \text{if } x' \text{ fresh} \]

\[ \begin{align*}
\langle \{ \text{Send } x \ y \} T \rangle & \quad \langle T \rangle \\
\quad x = \xi \parallel \xi : Q & \quad \quad x = \xi \parallel \xi : y; Q
\end{align*} \]

\[ \text{R:rcv} \quad \langle \text{let } x = \{ \text{Receive } y \} \text{ in } S \text{ end } T \rangle \quad \langle S \{ x' / x \} T \rangle \quad \text{if } x' \text{ fresh} \]

\[ \begin{align*}
y = \xi \parallel \xi : Q; z \parallel z = w & \quad \quad y = \xi \parallel \xi : Q \parallel z = w \quad \text{if } x' = w
\end{align*} \]
unique thread identifiers
threads endowed with a history
syntactic delimiters to statements, to delimit their scope
queues with histories
Making let reversible

\[
\begin{array}{l}
\frac{t[H]\langle \text{let } x = v \text{ in } S \text{ end } T \rangle}{0} \quad \frac{t[H \ast x']\langle S\{x'/x\} \langle \text{esc } T \rangle \rangle}{x' = v} \quad \text{if } x' \text{ fresh}
\end{array}
\]

- unique thread id and past history
- history include the new action
- scope delimiter
Making let reversible

\[
\begin{array}{c}
\frac{t[H]\langle \text{let } x = v \text{ in } S \text{ end } T \rangle}{0} \quad \frac{t[H \ast x']\langle S\{x'/x\} \langle \text{esc } T \rangle \rangle}{x' = v} \quad \text{if } x' \text{ fresh}
\end{array}
\]

- unique thread id and past history
- history include the new action
- scope delimiter
Making let reversible

\[
\begin{align*}
&\quad \quad \quad t[H] \langle \text{let } x = v \text{ in } S \text{ end } T \rangle \\
0 &\quad | \\
&\quad t[H \ast x'] \langle S \{x'/x\} \langle \text{esc } T \rangle \rangle & \text{if } x' \text{ fresh}
\end{align*}
\]

- unique thread id and past history
- history include the new action
- scope delimiter
Making let reversible

\[
\begin{align*}
t[H]\langle \text{let } x = v \text{ in } S \text{ end } T \rangle & \quad 0 \\
\text{if } x' \text{ fresh} & \quad t[H \ast x']\langle S\{x'/x\} \langle \text{esc } T \rangle \rangle \quad x' = v
\end{align*}
\]

- unique thread id and past history
- history include the new action
- scope delimiter
Some rules

<table>
<thead>
<tr>
<th>snd</th>
<th>$t[H] \langle { \text{Send } x \ y } \ C \rangle$</th>
<th>$t[H] \uparrow x \ C$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x = \xi \parallel \xi : K</td>
<td>K_h$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rcv</th>
<th>$t[H] \langle \text{let } y = { \text{Receive } x } \text{ in } S \text{ end } C \rangle$</th>
<th>$t[H] \downarrow x(y') \langle S { y'/y } \langle \text{esc } C \rangle \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\theta \parallel \xi : K; t':z</td>
<td>K_h$</td>
</tr>
<tr>
<td></td>
<td>if $y'$ fresh $\land \theta \triangleq x = \xi \parallel z = w$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>snd$^{-1}$</th>
<th>$t[H] \uparrow x \ C$</th>
<th>$t[H] \langle { \text{Send } x \ y } \ C \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x = \xi \parallel \xi : t:y; K</td>
<td>K_h$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rcv$^{-1}$</th>
<th>$t[H] \downarrow x(z) \langle S \langle \text{esc } C \rangle \rangle$</th>
<th>$t[H] \langle \text{let } z = { \text{Receive } x } \text{ in } S \text{ end } C \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$z = w \parallel x = \xi \parallel \xi : K</td>
<td>t':y,t; K_h$</td>
</tr>
</tbody>
</table>
Roadmap

1. μOz
2. Reversible Debuggers
3. μOz reversible debugger
Motivations

From Omniscent Debugger
- debugging is easier if you can go backward.
- no “Whoops, I went too far” while debugging with breakpoints
- no *guessing* where to put breakpoints

From UndoSoftware
Reversible debugging (also known as replay or historical debugging) allows a developer to step or run an application backwards, and so quickly track down the root-cause of even the most difficult bugs.
Three main techniques for reversible debugging:

- **Program instrumentation**
  - ad-hoc function are added to the source code in order to revert it
  - instrumentation can be enabled/disabled for space reason
  - the programmer decides which code section to instrument = guessing

- **Replay** [Bidirectional debugging]
  - instead of undoing the last $n$ steps, the program is re-executed till a point equivalent to going back of $n$ steps

- **Checkpoint + replay** [Igor]
  - periodically a checkpoint of the entire program is taken
  - restores a checkpoint + executes missing steps
Cons of existing techniques

- Usually is given to the user to \textit{trim} the debugger
  - which portion of code to record/monitor?
  - what size of the buffer to use?
- In multi-threaded system the execution is always the same (\textit{global order} among actions)
- No causally consistent backward execution

What about using an interpreter of a reversible language?

- program instrumentation “for free”
- causally consistent
Roadmap

1. μOz

2. Reversible Debuggers

3. μOz reversible debugger
μOz reversible debugger

- Java based interpreter of both μOz forward and backward semantics
- allows to roll-back a thread of n steps à la roll-π
  - causing the rollback of other threads
Example of execution

let $a = \text{true}$ in
\hspace{1cm} \text{(1)}

let $b = \text{false}$ in \hspace{1cm} \text{(2)}

let $x = \text{port}$ in \hspace{1cm} \text{(3)}

\hspace{1cm} \text{thread} \{ \text{send } x \ a \}; \text{skip}; \{ \text{send } x \ b \} \ \text{end}; \hspace{1cm} \text{(4)}

\hspace{1cm} \text{let} \ y = \{ \text{receive } x \} \ \text{in} \ \text{skip} \ \text{end} \hspace{1cm} \text{(5)}

\hspace{1cm} \text{end} \hspace{1cm} \text{(6)}

\hspace{1cm} \text{end} \hspace{1cm} \text{(7)}

\hspace{1cm} \text{end} \hspace{1cm} \text{(8)}

- at line (4) thread $t_1$ is created from thread $t_0$
- $t_1$ fully executes, then $t_0$ fully executes
- what should be the shape of $t_0$ (and of the port) if $t_1$ rolls of 3 steps?
Desired execution

\[
\begin{align*}
t_0 & \quad \text{let } y = \{\text{receive } x\} \text{ in skip end} \\
t_1 & \quad \{\text{send } x \ a\}; \text{skip}; \{\text{send } x \ b\} \\
x & \quad \bot
\end{align*}
\]

- \(t_0\) is rolled-back enough in order to \textbf{free} the read value
- No domino effect, causing \(t_0\) to fully roll-back
Causal dependencies

- send, receive and spawning operations create dependencies among threads
- sending on a channel makes values already present on it depending on the send (FIFO queues)
- reading from on a channel makes previous reads causally dependent on it (LIFO history)
- reading a value from a channel makes the reader causally dependent from the sender
Example: reversing a send

The red block depends on the pink ones and the blue ones.
int reverse (⋯)
{
    ⋯
    if(history.get(t_name).isSend())
        if(chan.isEmpty())
            return reverse(chan.getReaders() U t_name)
        if(!chan.getValue().isMine(t_name))
            return reverse(chan.getSenders() U t_name)
    // code to consume the msg from the channel
    ⋯
}
Channels contain also the \#inst (similar to roll-$\pi$ gammas) of I/O operations

- \#inst are unique
- total order on \#inst of the same thread (partial among threads)
- act like pc

in chan history instead of $(t_0, a, t_1)$ we have $(t_0, i, a, t_1, j)$

meaning that the $i$-th instruction of $t_0$ has sent $a$ that has been consumed by the $j$-th instruction of $t_1$
Reversing a send: code snippet

```java
if(ch.isEmpty())
    throw new WrongElementChannel(.., ch.getReaders(thread_id));
IValue val = ch.reverseSend(thread_id);
if(val == null)
{
    throw new WrongElementChannel(.., ch.getSenders(thread_id));
}
if(val.getType() == ValueType.ID){
    // reverse the action
}

getReaders and getSenders return a list of pair (thread_id, i) with i being the least instruction to which a thread should get back.
```
private static void rollTill(HashMap<String, Integer> map) {
    Iterator<String> it = map.keySet().iterator();
    while (it.hasNext()) {
        String id = it.next();
        int gamma = map.get(id);
        while (true) {
            try {
                int nro = stepBack(id);
                if (nro == gamma) { // reversed thread till the right gamma
                    break;
                }
            } catch (WrongElementChannel e) {
                rollTill(e.getDependencies());
            } catch (ChildMissingException e) {
                rollEnd(e.getChild()); // generated if a child has not empty history
            }
        }
    }
}
Future work

- improve the language
  - more data types
  - more constructs
  - add reversible pattern-matching [Yokoyama et al.]

- improve the debugger
  - watch-points and breakpoints
  - GUI

- study reversible jellyfishes

Future work

- improve the language
  - more data types
  - more constructs
  - add reversible pattern-matching [Yokoyama et al.]

- improve the debugger
  - watch-points and breakpoints
  - GUI

- study reversible jellyfishes

Future work

- improve the language
  - more data types
  - more constructs
  - add reversible pattern-matching [Yokoyama et al.]

- improve the debugger
  - watch-points and breakpoints
  - GUI

- study reversible jellyfishes