A reversible debugger for μOz

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

Syntax

$S ::=$

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

$$v ::= \textbf{true} \mid \textbf{false}$$

$$c ::= \textbf{proc}\ \{\tilde{x}\}\ S\ \textbf{end}$$

μOz semantics

- programs written as stacks of instructions
- a rule transforms a pair (program,state) into a new pair
- variables are **always** created fresh and never **modified**
- sent values are variables names, not their contents

μOz sample rules

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

Reversing μOz

- unique thread identifiers
- threads endowed with a history
- syntactic delimiters to statements, to delimit their scope
- queues with histories

Making let reversible

$$\frac{t[H]\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \;\Big\|\; \frac{t[H * x']\langle S\{^{x'}/_{x}\}\ \langle \textbf{esc } T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

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

Making let reversible

$$\frac{t[H]\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \;\Big\|\; \frac{t[H * x']\langle S\{^{x'}/_{x}\}\ \langle \textbf{esc } T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

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

Making let reversible

$$\frac{t[H]\langle \textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \quad \Big\| \quad \frac{t[H \ * x']\langle S\{^{x'}/_{x}\}\ \langle \textsf{esc}\ T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

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

Making let reversible

$$\frac{t[H]\langle\textbf{let } x = v \textbf{ in } S \textbf{ end } T\rangle}{0} \;\Big\|\; \frac{t[H * x']\langle S\{^{x'}/_{x}\}\ \langle\textbf{esc } T\rangle\rangle}{x' = v} \text{ if } x' \text{ fresh}$$

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

Some rules

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

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

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

snd^{-1} $$\frac{t[H\ \ \uparrow x]C}{x=\xi \parallel \xi : t{:}y;K|K_h} \Bigg\| \frac{t[H]\langle\{\ \texttt{Send}\ x\ y\ \}\ C\rangle}{x=\xi \parallel \xi : K|K_h}$$

rcv^{-1} $$\frac{t[H\ \ \downarrow x(z)]\langle S\ \langle \textbf{esc}\ C\rangle\rangle}{z=w \parallel x=\xi \parallel \xi : K|t'{:}y,t;K_h} \Bigg\| \frac{t[H]\langle \textbf{let}\ z = \{\ \texttt{Receive}\ x\ \}\ \textbf{in}\ S\ \textbf{end}\ C\rangle}{x=\xi \parallel \xi : K;t'{:}y|K_h}$$

Roadmap

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.

Techniques

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 *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 (**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

$$
\begin{array}{ll}
\textbf{let } a = \textbf{true} \;\; \textbf{in} & (1)\\
\quad \textbf{let } b = \textbf{false} \;\; \textbf{in} & (2)\\
\quad\quad \textbf{let } x = \textbf{port in} & (3)\\
\quad\quad\quad \textbf{thread } \{\textbf{send x a}\}; \textbf{skip}; \{\textbf{send x b}\} \textbf{ end}; & (4)\\
\quad\quad\quad \textbf{let } y = \{\textbf{receive } x\} \textbf{ in skip end} & (5)\\
\quad\quad \textbf{end} & (6)\\
\quad \textbf{end} & (7)\\
\textbf{end} & (8)
\end{array}
$$

- 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

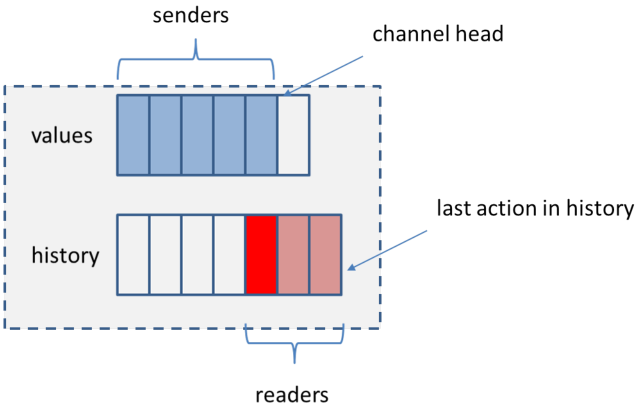
$$
\begin{array}{ll}
t_0 & \textbf{let } y = \{\textbf{receive } x\} \textbf{ in skip end} \\
t_1 & \{\textbf{send x a}\}; \textbf{skip}; \{\textbf{send x b}\} \\
x & \perp
\end{array}
$$

- t_0 is rolled-back enough in order to 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

Reverse Send

```
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
...
}
```

how do we know till when to reverse a dependant thread?

Implementation tricks

Channels contain also the #**inst** (similar to roll-π 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

```
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 $(\text{thread\_id}, i)$ with i being the least instruction to which a thread should get back.

Reversing: code snippet

```
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

1. http://proton.inrialpes.fr/~mezzina/deb/
2. https://code.google.com/p/moz-reversible-debugger/

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

1. http://proton.inrialpes.fr/~mezzina/deb/
2. https://code.google.com/p/moz-reversible-debugger/

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

1. http://proton.inrialpes.fr/~mezzina/deb/
2. https://code.google.com/p/moz-reversible-debugger/