

Modular rollback through free monads

Conor McBride, Olin Shivers, **Aaron Turon**

Goals for this talk

- Illustrate modular rollback with *side-effects*
- Brief tutorial on:
 - free monads
 - Filinski's monadic representation
- Derive modular rollback from these tools
- **Non-goal:** showing significant new research

DEMO

```
datatype sexp
  = Atom of string
  | Cons of sexp * sexp
  | Nil
```

```
val peek: unit -> char
val nom: unit -> unit
val abort: unit -> 'a
```

```
(* sexp: unit -> SExp *)  
fun sexp() =  
  case peek()  
  of #"." => abort()  
   | #")" => abort()  
   | #"(" => (nom(); openP())  
   | #" " => (nom(); sexp())  
   | _    => Atom (atom())
```

We use **effectful** operations:

```
val peek:    unit -> char  
val nom:    unit -> unit  
val abort:  unit -> 'a
```

Step 1: parameterize

```
signature READLINE =  
sig  
  val peek:    unit -> char  
  val nom:     unit -> unit  
  val abort:   unit -> 'a  
end
```

Step 1: parameterize

```
signature READLINE =
```

```
sig
```

```
  val peek: unit -> char
```

```
  val nom: unit -> unit
```

```
  val abort: unit -> 'a
```

```
end
```

```
functor Parser(R: READLINE) = ...
```

```
  val parse: unit -> sexp
```

A monad has:

return

bind

effectful operations

Free monads

treat these *syntactically*

peek: unit -> char
nom: unit -> unit
abort: unit -> α

peek:	char	readline
nom:	unit	readline
abort:	α	readline

Peek:	char	readline
Nom:	unit	readline
Abort:	α	readline

Peek: char readline

Nom: unit readline

Abort: α readline

Return: $\alpha \rightarrow \alpha$ readline

Bind: α readline

$\rightarrow (\alpha \rightarrow \beta$ readline)

$\rightarrow \beta$ readline

```
(* atom: unit -> string *)  
fun atom() =  
  case peek()  
  of (#" " | #"." | #"(" | #")") => ""  
     | c => (nom(); c ^ atom())
```

```
(* atom: unit -> string *)  
fun atom() =  
  case peek()  
  of (#" " | #"." | #"(" | #")") => ""  
   | c => (nom(); c ^ atom())
```

```
(* atom: string readline *)
```

```
val atom =
```

```
  case peek()
```

```
  of (#" " | #"." | #"(" | #")") => ""
```

```
  | c => (nom(); c ^ atom())
```

```
(* atom: string readline *)
```

```
val atom = Bind Peek (fn c =>
```

```
  case c
```

```
    of (#" " | #"." | #"(" | #")") => ""
```

```
    | _ => (nom(); c ^ atom())
```

```
(* atom: string readline *)  
val atom = Bind Peek (fn c =>  
  case c  
  of (#" " | #".") | #"(" | #")") =>  
     Return ""  
  | _ => (nom(); c ^ atom()))
```

```
(* atom: string readline *)  
val atom = Bind Peek (fn c =>  
  case c  
  of (#" " | #"." | #"(" | #")") =>  
    Return ""  
  | _ => Bind Nom (fn () =>  
    Bind atom (fn s =>  
    Return (c ^ s))))
```

Reify delimits your monadic computation

```
reify:     $\alpha$  [readline] ->  $\alpha$  readline  
reify c = reset (fn () => Return c)
```

**Reflect: to do an effect, put a bind
between you and your continuation**

```
reflect:  $\alpha$  readline  $\rightarrow$   $\alpha$  [readline]  
reflect v = shift (fn k  $\Rightarrow$  Bind v k)
```

Step 2: monadic reflection

```
val peek:  unit -> char  
val nom:   unit -> unit  
val abort: unit -> 'a
```

Step 2: monadic reflection

```
fun peek()    = shift (fn k => Bind Peek k)
fun nom()    = shift (fn k => Bind Nom k)
fun abort()  = shift (fn k => Bind Abort k)
```

Step 2: monadic reflection

```
fun peek()    = shift (fn k => Bind Peek k)
fun nom()     = shift (fn k => Bind Nom k)
fun abort()  = shift (fn k => Bind Abort k)
```

```
val Parse: sexp readline
val Parse = reset parse
```

```

interp: 'a readline ->
  string -> ('a * string) option
fun interp (Return x) s = SOME(x, s)
| interp (Bind f k) s =
  case f s
  of SOME (a, s') => k a s'
  | NONE          => NONE
| interp Peek s =
  if isEmpty(s) then NONE
  else SOME(sub s 0, s)
| interp Nom s =
  if isEmpty(s) then NONE
  else SOME((), triml 1 s)
| interp Abort s = NONE

```

The TTY Monad

Putc: char \rightarrow unit tty

Getc: char tty

Return: $\alpha \rightarrow \alpha$ tty

Bind: α tty \rightarrow ($\alpha \rightarrow \beta$ tty)
 $\rightarrow \beta$ tty

Step 3: Monad Mapping

```
datatype  $\alpha$  stk
  = PEEK of ( $\alpha$  stk) (char ->  $\alpha$  readline)
  | NOM of  $\alpha$  stk
  | ROOT of  $\alpha$  readline

val driver:  $\alpha$  stk ->
   $\alpha$  readline ->
  char option ->
   $\alpha$  tty
```

driver stk (Return x) _ = TTY.Return x

```
driver stk (Return x) _ = TTY.Return x
driver stk (Bind Peek k) NONE =
  TTY.Bind Getc (fn c =>
    case c of
      #"\b" => pop stk
      _      => driver (PEEK stk k) (k c) (SOME c))
```

```

driver stk (Return x) _ = TTY.Return x
driver stk (Bind Peek k) NONE =
  TTY.Bind Getc (fn c =>
    case c of
      #"\b" => pop stk
      _      => driver (PEEK stk k) (k c) (SOME c))
driver stk (Bind Peek k) (SOME c) =
  driver stk (k c) (SOME c)

```

```

driver stk (Return x) _ = TTY.Return x
driver stk (Bind Peek k) NONE =
  TTY.Bind Getc (fn c =>
    case c of
      #"\b" => pop stk
      _      => driver (PEEK stk k) (k c) (SOME c))
driver stk (Bind Peek k) (SOME c) =
  driver stk (k c) (SOME c)
driver stk (Bind Nom k) (SOME c) =
  Putc c >> driver (NOM stk) (k ()) NONE

```

```

driver stk (Return x) _ = TTY.Return x
driver stk (Bind Peek k) NONE =
  TTY.Bind Getc (fn c =>
    case c of
      #"\b" => pop stk
      _      => driver (PEEK stk k) (k c) (SOME c))
driver stk (Bind Peek k) (SOME c) =
  driver stk (k c) (SOME c)
driver stk (Bind Nom k) (SOME c) =
  Putc c >> driver (NOM stk) (k ()) NONE
driver stk (Bind Abort _) _ = pop stk

```

```
val pop:  $\alpha$  Stk ->  $\alpha$  tty
```

```
pop (PEEK stk k) =  
  driver stk (Bind Peek k) NONE
```

```
pop (NOM stk) =  
  Putc #"\b" >> Putc #"" >>  
  Putc #"\b" >> pop stk
```

```
pop (ROOT r) =  
  driver (ROOT r) r NONE
```

Conclusion

- The shift/reset in the pearl is *just* monadic representation
- Filinski's technique is useful for sneaking in *unexpected* effects