Functional Programming

WS 2015/16

Cezary Kaliszyk (VO+PS)
Yann Savoye (PS)
Łukasz Czajka (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 14

Overview

- Week 14 - Monads
  - Summary of Week 13
  - Example Monads

Overview

- Week 14 - Monads
  - Summary of Week 13
  - Example Monads

Dependent Types

- Simple type theory corresponds to propositional logic
  - A proof of a proposition corresponds to a program of a type
- With dependent types
  - Predicate logic, Closer to Math
  - Epigram, Cayenne, Mizar
- Polymorphism
  - Limited in OCaml
- All possible dependencies
  - Foundation for Coq, Agda, Matita

Overview

- Week 14 - Monads
  - Summary of Week 13
  - Example Monads

This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction,
reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing,

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, dependent types, monads

Overview

- Week 14 - Monads
  - Summary of Week 13
  - Example Monads

Handling Exceptional Cases

Safe List Decomposition

- head of list

```
let safe_hd = function x::_ -> Some x
                     | _    -> None
```

Handling Exceptional Cases

Safe List Decomposition

- head of list

```
let safe_hd = function x::_ -> Some x
                     | _    -> None
```

- tail of list

```
let safe_tl = function _::xs -> Some xs
                     | _     -> None
```

Handling Exceptional Cases (cont'd)

Example

```
let div2 xs =
  match safe_hd xs with
    | None    -> None
    | Some x -> match safe_tl xs with
      | None     -> None
      | Some xs -> match safe_hd xs with
        | None    -> None
        | Some y -> if y = 0 then None
                              else Some(x/y)
```

Idea - Use HO-Functions to Handle Common Cases

Common Cases

- indicate an exceptional case

Idea - Use HO-Functions to Handle Common Cases

Common Cases

- indicate an exceptional case
- return a result wrapped in `Some`

Idea - Use HO-Functions to Handle Common Cases

Common Cases

- indicate an exceptional case
- return a result wrapped in `Some`
- check if previous result is `None` and act accordingly

Idea - Use HO-Functions to Handle Common Cases

Common Cases

- indicate an exceptional case
- return a result wrapped in `Some`
- check if previous result is `None` and act accordingly

HO-Functions - The Option Monad

Idea - Use HO-Functions to Handle Common Cases

Common Cases

- indicate an exceptional case
- return a result wrapped in `Some`
- check if previous result is `None` and act accordingly

HO-Functions - The Option Monad

- "error"

```
let error = None
```

Idea - Use HO-Functions to Handle Common Cases

Common Cases

- indicate an exceptional case
- return a result wrapped in `Some`
- check if previous result is `None` and act accordingly

HO-Functions - The Option Monad

- "error"

```
let error = None
```

- "return"

```
let return x = Some x
```

Idea - Use HO-Functions to Handle Common Cases

Common Cases

- indicate an exceptional case
- return a result wrapped in `Some`
- check if previous result is `None` and act accordingly

HO-Functions - The Option Monad

- "error"

```
let error = None
```

- "return"

```
let return x = Some x
```

- "bind"

```
let bind m f = match m with None   -> None
                          | Some x -> f x
```

Handling Exceptional Cases (cont'd)

Note

for convenience, we use an infix version of `bind`

```
let (>>=) = bind
```

Example

```
let div2' xs =
  safe_hd xs >>= fun x  ->
  safe_tl xs >>= fun xs ->
  safe_hd xs >>= fun y  ->
  if y = 0 then error
           else return(x/y)
```

Maintaining State

Fresh Names

- a name is fresh, if it has not been used before

```
let fresh x (xs,i) = match Lst.lookup x xs with
  | Some y -> (y,(xs,i))
  | None   -> (i,((x,i)::xs,i+1))
```

Maintaining State (cont'd)

Rename All Variables of a λ-Term Consistently

```
let rec freshify state = function
  | Var x    ->
     let (y,state') = fresh x state in
     (Var y,state')
  | App(s,t) ->
    let (s',state') = freshify state s in
    let (t',state'') = freshify state' t in
    (App(s',t'),state'')
  | Lam(x,t) ->
    let (y,state')  = fresh x state in
    let (t',state'') = freshify state' t in
    (Lam(y,t'),state'')
```

Again - Use HO-Functions to Handle Common Cases

Common Cases

- read/write state

Again - Use HO-Functions to Handle Common Cases

Common Cases

- read/write state
- return a result without changing the state

Again - Use HO-Functions to Handle Common Cases

Common Cases

- read/write state
- return a result without changing the state
- maintain the state during all computations

Again - Use HO-Functions to Handle Common Cases

Common Cases

- read/write state
- return a result without changing the state
- maintain the state during all computations

HO-Functions - The State Monad

```
let get = fun s -> (s,s)

let set s = fun _ -> ((),s)

let return x = fun s -> (x,s)

let bind m f = fun s ->
  let (x,s') = m s in
  (f x) s'
```

Maintaining State (cont'd)

Example (Explicit Handling of State)

```
let fresh' x =
  get >>= fun(xs,i) -> match Lst.lookup x xs with
  | Some y -> return y
  | None   -> set((x,i)::xs,i+1) >>= fun _ ->
              return i
```

Maintaining State (cont'd)

Example (Implicit Handling of State)

```
let rec freshify = function
  | Var x    ->
    fresh' x >>= fun y ->
    return(Var y)
  | App(s,t) ->
    freshify s >>= fun s' ->
    freshify t >>= fun t' ->
    return(App(s',t'))
  | Lam(x,t)  ->
    fresh' x    >>= fun y ->
    freshify t >>= fun t' ->
    return(Lam(y,t'))
```

Nondeterminism

All Possible Combinations

```
let cross xs ys = Lst.concat(
  Lst.map (fun x ->
    Lst.map (fun y -> (x,y)) ys
  ) xs
)
```

Once Again - Use HO-Functions to Handle Common Cases

Common Cases

- turn an element into a singleton list

Once Again - Use HO-Functions to Handle Common Cases

Common Cases

- turn an element into a singleton list
- combine the result lists of a computation into a single list

Once Again - Use HO-Functions to Handle Common Cases

Common Cases

- turn an element into a singleton list
- combine the result lists of a computation into a single list

HO-Functions - The State Monad

```
let return x = [x]

let bind xs f = Lst.concat(Lst.map f xs)
```

Nondeterminism (cont'd)

Example

```
let cross' xs ys =
  xs >>= fun x ->
  ys >>= fun y ->
  return(x,y)
```

Syntactic Sugar - Do/Perform Notation

Note

needs `camlp4` and `pa_monad.cmo`

```
open OptionMonad

let div xs = perform
  x  <-- safe_hd xs;
  xs <-- safe_tl xs;
  y  <-- safe_hd xs;
  if y = 0 then error
           else return(x/y)
```

Syntactic Sugar - Do/Perform Notation (cont'd)

Note

needs camlp4 and pa_monad.cmo

```
open ListMonad
let cross xs ys = perform
  x <-- xs;
  y <-- ys;
  return(x,y)
```

Combining Monads

- a single monad enriches a datatype with additional functionality, e.g., error-handling, state, I/O, nondeterminism, ...
- sometimes more than one of those functionalities is desired at the same time
- this can be done by nesting monads (cf. monad transformers)

Further Reading

- http://enfranchisedmind.com/blog/posts/a-monad-tutorial-for-ocaml/
- http://en.wikipedia.org/wiki/Monad_(functional_programming)
- But not http://en.wikipedia.org/wiki/Mondas!