PPL2020.06.29

SCHEME

Define the construct define-with-types, that is used to define a procedure with type constraints, both for the parameters and for the return value. The type constraints are the corresponding type predicates, e.g. number? to check if a value is a number.

If the type constraints are violated, an error should be issued.

E.g.

(define-with-types (add-to-char : integer? (x : integer?) (y : char?))

(+ x (char->integer y)))

defines a procedure called add-to-char, which takes an integer and a character, and returns an integer.

HASKELL

We want to implement a queue, i.e. a FIFO container with the two operations

enqueue and dequeue with the obvious meaning. A functional way of doing this is

based on the idea of using two lists, say L1 and L2, where the first one is used

for dequeuing (popping) and the second one is for enqueing (pushing) When

dequeing, if the first list is empty, we take the second one and put it in the

first, reversing it This last operation appears to be O(n), but suppose we

have n enqueues followed by n dequeues; the first dequeue takes time

proportional to n (reverse), but all the other dequeues take constant time.

This makes the operation O(1) amortised that is why it is acceptable in many

applications.

1) Define Queue and make it an instance of Eq

2) Define enqueue and dequeue, stating their types

HASKELL (ii)

Make Queue an instance of Functor and Foldable

HASKELL (iii)

Make Queue an instance of Applicative

ERLANG

Define a "functional" process buffer, called fuffer, that stores only one value and may receive messages only from its creator. fuffer can receive the following commands:

'set' to store a new value

'get' to obtain the current value

'apply F' to apply the function F to the stored value

'die' to end

'duplicate' to create (and return) an exact copy of itself

SOLUTIONS

(define-syntax define-with-types

(syntax-rules (:)

((\_ (f : tf (x1 : t1) ...) e1 ...)

(define (f x1 ...)

(if (and (t1 x1) ...)

(let ((res (begin

e1 ...)))

(if (tf res)

res

(error "bad return type")))

(error "bad input types"))))))

data Queue a = Queue [a] [a] deriving Show

to\_list (Queue x y) = x ++ reverse y

instance Eq a => Eq (Queue a) where

q1 == q2 = (to\_list q1) == (to\_list q2)

enqueue :: a -> Queue a -> Queue a

enqueue x (Queue pop push) = Queue pop (x:push)

dequeue :: Queue a -> (Maybe a, Queue a)

dequeue q@(Queue [] []) = (Nothing, q)

dequeue (Queue (x:xs) v) = (Just x, Queue xs v)

dequeue (Queue [] v) = dequeue (Queue (reverse v) [])

instance Functor Queue where

fmap f (Queue x y) = Queue (fmap f x) (fmap f y)

instance Foldable Queue where

foldr f z q = foldr f z $ to\_list q

q1 +++ (Queue x y) = Queue ((to\_list q1) ++ x) y

qconcat q = foldr (+++) (Queue [][]) q

instance Applicative Queue where

pure x = Queue [x] []

fs <\*> xs = qconcat $ fmap (\f -> fmap f xs) fs

fuffer(Data, PID) ->

receive

{set, PID, V} ->

fuffer(V, PID);

{get, PID} ->

PID!Data, fuffer(Data, PID);

{apply, PID, F} ->

fuffer(F(Data), PID);

{die, PID} -> ok;

{duplicate, PID} ->

PID ! spawn(?MODULE, fuffer, [Data, PID]),

fuffer(Data, PID)

end.

PPL 2020.07.17

Ex 1 - Scheme

Define the verbose construct for folding illustrated by the following

example:

(cobol-fold direction -> from 1 data 1 2 3 4 5 6

(exec

(displayln y)

(+ x y))

using x y)

This is a fold-right (->) with initial value 1 on the list (1 2 3 4 5

6), and the fold function is given in the "exec" part. Of course, <-

is used to select fold-left instead of right.

Ex 2 - Haskell

Define a data type that stores an m by n matrix as a list of lists by row.

After defining an appropriate data constructor, do the following:

1. Define a function `new' that takes as input two integers m and n

and a value `fill', and returns an m by n matrix whose elements are all equal to `fill'.

2. Define function `replace' such that, given a matrix m, the indices i, j of one of its elements,

and a new element, it returns a new matrix equal to m except for the element

in position i, j, which is replaced with the new one.

3. Define function `lookup', which returns the element in a given position

of a matrix.

4. Make the data type an instance of Functor and Foldable.

5. Make the data type an instance of Applicative.

In your implementation you can use the following functions:

splitAt :: Int -> [a] -> ([a], [a])

unzip :: [(a, b)] -> ([a], [b])

(!!) :: [a] -> Int -> a

Ex 3 - Erlang

Define a "broadcaster" process which answers to the following

commands:

- {spawn, L, V} creates a process for each element of L, passing its

initial parameter in V, where L is a list of names of functions

defined in the current module and V is their respective parameters (of

course it must be |L| = |V|);

- {send, V}, with V a list of values, sends to each respective process

created with the previous spawn command a message in V; e.g. {spawn,

[1,2,3]} will send 1 to the first process, 2 to the second, and 3 to

the third;

- stop is used to end the broadcaster, and to also stop every process

spawned by it.

SOLUTIONS

Ex 1

(define-syntax cobol-fold

(syntax-rules (direction -> <- data using from exec)

((\_ direction -> from i data d ... (exec e ... ) using x y)

(foldr (lambda (x y) e ...) i '(d ...)))

((\_ direction <- from i data d ... (exec e ... ) using x y)

(foldl (lambda (x y) e ...) i '(d ...)))))

Ex 2

newtype Matrix a = Matrix [[a]] deriving (Eq, Show)

new :: Int -> Int -> a -> Matrix a

new m n fill = Matrix [[fill | \_ <- [1..n]] | \_ <- [1..m]]

replace :: Int -> Int -> a -> Matrix a -> Matrix a

replace i j x (Matrix rows) = let (rowsHead, r:rowsTail) = splitAt i rows

(rHead, x':rTail) = splitAt j r

in Matrix $ rowsHead ++ ((rHead ++ (x:rTail)):rowsTail)

lookup :: Int -> Int -> Matrix a -> a

lookup i j (Matrix rows) = (rows !! i) !! j

instance Functor Matrix where

fmap f (Matrix rows) = Matrix $ map (\r -> map f r) rows

instance Foldable Matrix where

foldr f e (Matrix rows) = foldr (\r acc -> foldr f acc r) e rows

hConcat :: Matrix a -> Matrix a -> Matrix a

hConcat (Matrix []) m2 = m2

hConcat m1 (Matrix []) = m1

hConcat (Matrix (r1:r1s)) (Matrix (r2:r2s)) =

let (Matrix tail) = hConcat (Matrix r1s) (Matrix r2s)

in Matrix $ (r1 ++ r2) : tail

vConcat :: Matrix a -> Matrix a -> Matrix a

vConcat (Matrix rows1) (Matrix rows2) = Matrix $ rows1 ++ rows2

concatMapM :: (a -> Matrix b) -> Matrix a -> Matrix b

concatMapM f (Matrix rows) =

let empty = Matrix []

in foldl

(\acc r -> vConcat acc $ foldl (\acc x -> hConcat acc (f x)) empty r)

empty

rows

instance Applicative Matrix where

pure x = Matrix [[x]]

fs <\*> xs = concatMapM (\f -> fmap f xs) fs

Ex 3

broadcaster(Pids) ->

receive

{spawn, Fs, Vs} ->

FDs = lists:zip(Fs, Vs),

io:format("~p~n", [FDs]),

broadcaster([spawn\_link(?MODULE, F, V) || {F,V} <- FDs]);

{send, Vs} ->

FDs = lists:zip(Pids, Vs),

io:format("~p~n", [FDs]),

[ Pid ! V || {Pid, V} <- FDs];

stop ->

ok

end.

PPL 2020.07.17

Ex 1 - Scheme

Define the verbose construct for folding illustrated by the following

example:

(cobol-fold direction -> from 1 data 1 2 3 4 5 6

(exec

(displayln y)

(+ x y))

using x y)

This is a fold-right (->) with initial value 1 on the list (1 2 3 4 5

6), and the fold function is given in the "exec" part. Of course, <-

is used to select fold-left instead of right.

Ex 2 - Haskell

Define a data type that stores an m by n matrix as a list of lists by row.

After defining an appropriate data constructor, do the following:

1. Define a function `new' that takes as input two integers m and n

and a value `fill', and returns an m by n matrix whose elements are all equal to `fill'.

2. Define function `replace' such that, given a matrix m, the indices i, j of one of its elements,

and a new element, it returns a new matrix equal to m except for the element

in position i, j, which is replaced with the new one.

3. Define function `lookup', which returns the element in a given position

of a matrix.

4. Make the data type an instance of Functor and Foldable.

5. Make the data type an instance of Applicative.

In your implementation you can use the following functions:

splitAt :: Int -> [a] -> ([a], [a])

unzip :: [(a, b)] -> ([a], [b])

(!!) :: [a] -> Int -> a

Ex 3 - Erlang

Define a "broadcaster" process which answers to the following

commands:

- {spawn, L, V} creates a process for each element of L, passing its

initial parameter in V, where L is a list of names of functions

defined in the current module and V is their respective parameters (of

course it must be |L| = |V|);

- {send, V}, with V a list of values, sends to each respective process

created with the previous spawn command a message in V; e.g. {spawn,

[1,2,3]} will send 1 to the first process, 2 to the second, and 3 to

the third;

- stop is used to end the broadcaster, and to also stop every process

spawned by it.

SOLUTIONS

Ex 1

(define-syntax cobol-fold

(syntax-rules (direction -> <- data using from exec)

((\_ direction -> from i data d ... (exec e ... ) using x y)

(foldr (lambda (x y) e ...) i '(d ...)))

((\_ direction <- from i data d ... (exec e ... ) using x y)

(foldl (lambda (x y) e ...) i '(d ...)))))

Ex 2

newtype Matrix a = Matrix [[a]] deriving (Eq, Show)

new :: Int -> Int -> a -> Matrix a

new m n fill = Matrix [[fill | \_ <- [1..n]] | \_ <- [1..m]]

replace :: Int -> Int -> a -> Matrix a -> Matrix a

replace i j x (Matrix rows) = let (rowsHead, r:rowsTail) = splitAt i rows

(rHead, x':rTail) = splitAt j r

in Matrix $ rowsHead ++ ((rHead ++ (x:rTail)):rowsTail)

lookup :: Int -> Int -> Matrix a -> a

lookup i j (Matrix rows) = (rows !! i) !! j

instance Functor Matrix where

fmap f (Matrix rows) = Matrix $ map (\r -> map f r) rows

instance Foldable Matrix where

foldr f e (Matrix rows) = foldr (\r acc -> foldr f acc r) e rows

hConcat :: Matrix a -> Matrix a -> Matrix a

hConcat (Matrix []) m2 = m2

hConcat m1 (Matrix []) = m1

hConcat (Matrix (r1:r1s)) (Matrix (r2:r2s)) =

let (Matrix tail) = hConcat (Matrix r1s) (Matrix r2s)

in Matrix $ (r1 ++ r2) : tail

vConcat :: Matrix a -> Matrix a -> Matrix a

vConcat (Matrix rows1) (Matrix rows2) = Matrix $ rows1 ++ rows2

concatMapM :: (a -> Matrix b) -> Matrix a -> Matrix b

concatMapM f (Matrix rows) =

let empty = Matrix []

in foldl

(\acc r -> vConcat acc $ foldl (\acc x -> hConcat acc (f x)) empty r)

empty

rows

instance Applicative Matrix where

pure x = Matrix [[x]]

fs <\*> xs = concatMapM (\f -> fmap f xs) fs

Ex 3

broadcaster(Pids) ->

receive

{spawn, Fs, Vs} ->

FDs = lists:zip(Fs, Vs),

io:format("~p~n", [FDs]),

broadcaster([spawn\_link(?MODULE, F, V) || {F,V} <- FDs]);

{send, Vs} ->

FDs = lists:zip(Pids, Vs),

io:format("~p~n", [FDs]),

[ Pid ! V || {Pid, V} <- FDs];

stop ->

ok

end.

PPL20210120

SCHEME

Define a pure function (i.e. without using procedures with side effects, such as set!) which takes a multi-level list, i.e. a list that may contain any level of lists, and converts it into a data structure where each list is converted into a vector.

E.g.

The result of (multi-list->vector '(1 2 (3 4) (5 (6)) "hi" ((3) 4))))

should be: '#(1 2 #(3 4) #(5 #(6)) "hi" #(#(3) 4))

HASKELL

Consider the following data structure for general binary trees:

data Tree a = Empty | Branch (Tree a) a (Tree a) deriving (Show, Eq)

Using the State monad as seen in class:

1) Define a monadic map for Tree, called mapTreeM.

2) Use mapTreeM to define a function which takes a tree and returns a tree containing list of elements that are all the data found in the original tree in a depth-first visit.

E.g.

From the tree: (Branch (Branch Empty 1 Empty) 2 (Branch (Branch Empty 3 Empty) 4 Empty))

we obtain:

Branch (Branch Empty [1] Empty) [1,2] (Branch (Branch Empty [1,2,3] Empty) [1,2,3,4] Empty)

ERLANG

Define a function for a proxy used to avoid to send PIDs; the proxy must react to the following messages:

- {remember, PID, Name}: associate the value Name with PID.

- {question, Name, Data}: send a question message containing Data to the PID corresponding to the value Name (e.g. an atom), like in PID ! {question, Data}

- {answer, Name, Data}: send an answer message containing Data to the PID corresponding to the value Name (e.g. an atom), like in PID ! {answer, Data}

SOLUTIONS

1)

(define (multi-list->vector lst)

(cond

((not (list? lst)) lst)

((null? (filter list? lst)) (apply vector lst))

(else (apply vector (map multi-list->vector lst)))))

2)

mapTreeM :: Monad m => (t -> m a) -> Tree t -> m (Tree a)

mapTreeM f Empty = return Empty

mapTreeM f (Branch lhs v rhs) = do

lhs' <- mapTreeM f lhs

v1 <- f v

rhs' <- mapTreeM f rhs

return (Branch lhs' v1 rhs')

depth\_tree t = let (State f) = mapTreeM

(\v -> do cur <- getState

putState $ cur ++ [v]

getState)

t

in snd $ f []

3)

proxy(Table) ->

receive

{question, Name, Data} ->

#{Name := Id} = Table,

Id ! {question, Data},

proxy(Table);

{answer, Name, Data} ->

#{Name := Id} = Table,

Id ! {answer, Data},

proxy(Table);

{remember, PID, Name} ->

proxy(Table#{Name => PID})

end.

PPL 2021.02.08

Ex 1

SCHEME:

Write a function 'depth-encode' that takes in input a list possibly containing

other lists at multiple nesting levels, and returns it as a flat list where

each element is paired with its nesting level in the original list.

E.g. (depth-encode '(1 (2 3) 4 (((5) 6 (7)) 8) 9 (((10)))))

returns

((0 . 1) (1 . 2) (1 . 3) (0 . 4) (3 . 5) (2 . 6) (3 . 7) (1 . 8) (0 . 9) (3 . 10))

Ex 2

HASKELL:

A multi-valued map (Multimap) is a data structure that associates keys of

a type k to zero or more values of type v. A Multimap can be represented as

a list of 'Multinodes', as defined below. Each multinode contains a unique key

and a non-empty list of values associated to it.

data Multinode k v = Multinode { key :: k

, values :: [v]

}

data Multimap k v = Multimap [Multinode k v]

1) Implement the following functions that manipulate a Multimap:

insert :: Eq k => k -> v -> Multimap k v -> Multimap k v

insert key val m returns a new Multimap identical to m, except val is added to the values associated to k.

lookup :: Eq k => k -> Multimap k v -> [v]

lookup key m returns the list of values associated to key in m

remove :: Eq v => v -> Multimap k v -> Multimap k v

remove val m returns a new Multimap identical to m, but without all values equal to val

2) Make Multimap k an instance of Functor.

Ex 3

ERLANG:

Consider the apply operation (i.e.<\*>) in Haskell's Applicative class.

Define a parallel <\*> for Erlang's lists.

Solutions

Ex 1

(define (depth-encode ls)

(define (enc-aux l)

(cond ((null? l) l)

((list? (car l))

(append (map (λ (nx) (cons (+ (car nx) 1) (cdr nx)))

(enc-aux (car l)))

(enc-aux (cdr l))))

(else (cons (cons 0 (car l)) (enc-aux (cdr l))))))

(enc-aux ls))

Ex 2

empty :: Multimap k v

empty = Multimap []

insert :: Eq k => k -> v -> Multimap k v -> Multimap k v

insert key val (Multimap []) = Multimap [Multinode key [val]]

insert key val (Multimap (m@(Multinode nk nvals):ms))

| nk == key = Multimap ((Multinode nk (val:nvals)):ms)

| otherwise = let Multimap p = insert key val (Multimap ms)

in Multimap (m:p)

lookup :: Eq k => k -> Multimap k v -> [v]

lookup \_ (Multimap []) = []

lookup key (Multimap ((Multinode nk nvals):ms))

| nk == key = nvals

| otherwise = lookup key (Multimap ms)

remove :: Eq v => v -> Multimap k v -> Multimap k v

remove val (Multimap ms) = Multimap $ foldr mapfilter [] ms

where mapfilter (Multinode nk nvals) rest =

let filtered = filter (/= val) nvals

in if null filtered

then rest

else (Multinode nk filtered):rest

instance Functor (Multimap k) where

fmap f (Multimap m) = Multimap (fmap (mapNode f) m) where

mapNode f (Multinode k v) = Multinode k (fmap f v)

Ex 3

runit(Proc, F, X) ->

Proc ! {self(), F(X)}.

pmap(F, L) ->

W = lists:map(fun(X) ->

spawn(?MODULE, runit, [self(), F, X])

end, L),

lists:map(fun (P) ->

receive

{P, V} -> V

end

end, W).

pappl(FL, L) ->

lists:foldl(fun (X,Y) -> Y ++ X end, [], pmap(fun(F) -> pmap(F, L) end, FL)).