Principles of Programming Languages

2015.09.10

Notes

- NAME: _____
- Did you present a small project? YES / NO
- Total available time: 1h 30'.
- You may use any written material you need.
- You cannot use computers or phones during the exam.

1 Scheme

Consider the following procedure:

```
(define (re-map f L cond?)
  (let loop ((res '())
             (cur L))
    (if (null? cur)
      res
      (let* ((k #f)
             (v (call/cc
                  (lambda (cont)
                    (set! k cont)
                    (f (car cur))))))
        (if (cond? v)
          (cons k v)
          (loop (append res (list v))
                (cdr cur)))))))
```

1.1 Description (5 points)

Give a brief explanation of what `re-map` does, providing also a simple but meaningful example of its use and return value, different from the one of the next question.

1.2 Example usage (5 points)

Let us consider to use `re-map` at the REPL with the following command:

```
> (define V
   (re-map (lambda (x) (+ x 1))
           '(0 1 -4 3 -6 5)
           negative?))
```

Give a sequence of related commands such that the result of the last command is the list `(1 2 3 4 5 6)`.

2 Haskell

2.1 Class definition (3 points)

Define a class called $Blup$, for a generic type T having two parameters x and y, providing two operations called $fisto$ and $fosto$. $fisto$ takes a value belonging to T and returns a value of type $Maybe\ x$, while $fosto$ takes a value belonging to T and returns a value of type $Maybe\ y$.

2.2 Instance I (4 points)

Define the sum type $Blargh$ with two parameters of types a and b. It has three data constructor: either Bip with two parameters of types respectively a and b, or Bop with only one parameter of type a, or Bup with no parameters.

Make $Blargh$ an instance of class $Blup$, where $fisto$ is used to access to data of type a, and $fosto$ to data of type b.

2.3 Instance II (4 points)

Define the sum type $Blarf$ with two parameters of types a and b. It has two data constructor: either La and a list of elements of type a, or Lb and a list of elements of type b.

Make $Blarf$ an instance of class $Blup$, where $fisto$ is used to access to the head of the list of elements of type a, and $fosto$ to the head of the list of elements of type b.

2.4 Smap (6 points)

1. Define a function $smap$ that takes an infinite list L of Int, a function f from Int to Int, an operation OP over Int, and a threshold T. $smap$ performs a map of f on L, while keeping an accumulator K (with starting value 0), which is updated at each step as $oldAccumulatorValue\ OP\ f(currentElementOfL)$. $smap$ stops when the value of K reaches T and returns a list of all the computed values of the map. E.g. `smap (^2) (+) [1,2..] 100` is the list [1,4,9,16,25,36,49].
2. Write $smap$'s type.

3 Prolog (5 points)

Define a "triparting" predicate that, given a list L and two pivot values, returns three lists such that the first contains all the values of L less than both pivots, the second contains values between the pivots (including the extremes), the last contains all the remaining values.

Solutions

Scheme

The procedure `re-map` is a map with a predicate condition (parameter `cond?`). If the condition holds for the current computed value v of the map, `re-map` returns a pair holding the continuation and v. It is possible to continue its computation, by providing a substitute value to the returned continuation, and calling it. Hence, we can obtain the requested list by performing:

```
> ((car V) (abs (cdr V)))  ; we could just use 3
> ((car V) (abs (cdr V)))  ; idem with 5
> V
```

Haskell

```
class Blup a where
    fisto :: (a b c) -> Maybe b
    fosto :: (a b c) -> Maybe c

data Blargh a b = Bip a b | Bop a | Bup deriving (Show, Eq)

instance Blup Blargh where
    fisto (Bip a b) = Just a
    fisto (Bop a) = Just a
    fisto Bup = Nothing

    fosto (Bip a b) = Just b
    fosto _ = Nothing


data Blarf a b = La [a] | Lb [b] deriving (Show, Eq)

instance Blup Blarf where
    fisto (La (x:xs)) = Just x
    fisto _ = Nothing

    fosto (Lb (x:xs)) = Just x
    fosto _ = Nothing


smap :: (Int -> Int) -> (Int -> Int -> Int) -> [Int] -> Int -> [Int]
smap f op list end = smapp f op list 0 [] end

smapp f op (x:xs) acc res end | acc >= end = res
smapp f op (x:xs) acc res end = smapp f op xs (op acc v) (res ++ [v]) end
                                    where v = f x
```

Prolog

It is a simple variant of Quicksort's partition as seen in class. Must be called with $P1 < P2$.

```
tripart([X|L],P1,P2,[X|L1],L2,L3) :- X < P1,  X < P2,  !, tripart(L,P1,P2,L1,L2,L3).
tripart([X|L],P1,P2,L1,[X|L2],L3) :- X >= P1, X =< P2, !, tripart(L,P1,P2,L1,L2,L3).
tripart([X|L],P1,P2,L1,L2,[X|L3]) :- X > P1,  X > P2,  !, tripart(L,P1,P2,L1,L2,L3).
tripart([],_,_,[],[],[]).
```