Purely Functional Algorithm Specification

Jan van Eijck

SEN1 Talk, Amsterdam, June 30 2011

Abstract

An algorithm is an effective method expressed as a list of instructions describing a computation for calculating a result. Algorithms have to be written in human readable form, either using pseudocode (natural language looking like executable code), a high level specification language like Dijkstra's Guarded Command Language, or an executable formal specification formalism such as Z.

We will give an easy purely functional implementation of the Guarded Command Language, and demonstrate how this can be used for specifying (executable) algorithms, and for automated testing of Hoare correctness statements about these algorithms. The small extension of Haskell that we will present and discuss can be viewed as a domain specific language for algorithm specification and testing.

Inspiration for this was the recent talk by Leslie Lamport on the executable algorithm specification language PlusCal [5], and Edsger W. Dijkstra, ”EWD472: Guarded commands, non-determinacy and formal derivation of programs.” [3] Instead of formal program derivation, we demonstrate test automation of Hoare style assertions.

Overview

We show how to write imperative algorithms in a purely functional style. No monads needed.

Our method yields executable algorithm specifications together with executable tests, so the specifications can be easily tested and debugged.

We will first explain algorithm specification and Hoare assertion over algorithm execution, by explaining how the algorithm construction operators are defined.

Next, we illustrate by means of several example algorithms: squaring a natural number in terms of addition, Euclid's gcd algorithm and the extended gcd algorithm, and Prim's algorithm for finding a minimum spanning tree of a connected undirected weighted graph.

Module Declaration

Import modules for sets and lists, for turning properties into tests, and for injection of randomness into the function space.

```
module GCL where

import Data.Set as Set
import Data.List as List
import Test.QuickCheck
import System.Random
import System.IO.Unsafe
```

Variable Names, Values, Counters

Variable names are strings.

```
type Name = String
```

Values can be anything.

We will let the type of values be a parameter of states.

State Spaces

A state is a triple consisting of a state counter, a variable space, and a space for sets.

Because we want a record of everything we will carry along the complete state space.

Because we want to assign to sets, we throw in space for them too. Single item variables point to functions from `Int` lists to values. This will take care of array assignment: arrays are functions, really.

```
type Counter = Integer
type Space = Counter -> Name -> [Int] -> Int
type SetSpace = Counter -> Name -> Set Int
type State = (Counter, Space, SetSpace)
```

Non-Determinism

For nondeterministic choice (selection) we need randomization. The following function generates a random natural number less than a given n.

```
randomNat :: Int -> Int
randomNat n = unsafePerformIO $
              getStdRandom (randomR (0,n-1))
```

Importing of the random number through IO cannot be avoided: we have to get the random seed from the environment somehow.

Picking Randomly from a List

```
pick :: [a] -> (Int,a)
pick []  = error "pick from empty list"
pick [x] = (0,x)
pick xs  = (n, xs !! n)
  where n = randomNat (length xs)
```

Removing randomly from a list:

```
pickRemove :: [a] -> (a,[a])
pickRemove xs = (x,xs') where
   (n,x)   = pick xs
   xs' = take n xs ++ drop (n+1) xs
```

Operations on Sets

Taking an arbitrary element from a (non-empty) set.

```
takeElem :: Set a -> a
takeElem = snd . pick . elems
```

Note that this is a non-deterministic operation.

Taking an arbitrary element from a (non-empty) set and removing it.

```
takeRemove :: Ord a => Set a -> (a,Set a)
takeRemove set = (x,set')
  where x    = takeElem set
        set' = Set.delete x set
```

Note that this is a non-deterministic operation.

Intensions of Values, Conditions

Intensions of values of type `a` are functions from states to values. We abbreviate this type as `I a` , or `SI a` for sets.

```
type I a = State -> a
type SI a = State -> Set a
```

Intensions of values will be used for variable assignment (see below).

A condition is the intension of a Boolean: a function from states to Booleans.

```
type Condition = State -> Bool
```

Lifting Numeric Operations to Intensions of Numerals

```
infixl 6 <+>

(<+>) :: Num a => I a -> I a -> I a
x <+> y = \s -> x s + y s

infixl 6 <->

(<->) :: Num a => I a -> I a -> I a
x <-> y = \s -> x s - y s

infixr 7 <*>

(<*>) :: Num a => I a -> I a -> I a
x <*> y = \s -> x s * y s
```

Actions

An `Action` is what corresponds to an imperative statement.

Variable assignments, e.g., are (interpreted as) actions.

An action is a state transformation, i.e., a function from states to states.

```
type Action = State -> State
```

Single Variable Lookup

Since a state consists of a counter and a space, to find the value of a variable name we have to apply the space to the counter and the name. For single variable lookup, we apply to the empty list.

```
infixl 9 @@

(@@) :: State -> Name -> Int
(c,f,_) @@ x = f c x []
```

The empty list is there because single variable lookup is a special case of array lookup, with an empty index list.

Note that `(@@"x")` is the function that maps a state `s` to the value of `"x"` in `s`.

Array Variable Lookup

An array is a function from index lists to values. Array variable lookup retrieves the appropriate function.

```
infixl 9 @|

(@|) :: State -> (Name,[Int]) -> Int
(c,f,_) @| (x,is) = f c x is
```

Note that for any state `s` it holds that `s@|("x",[])` is equal to `s@@"x"`.

Variable Lookup for Sets

```
infixl @@@

(@@@) :: State -> Name -> Set Int
(c,_,s) @@@ x = s c x
```

Start State

Construct a suitable start state by initializing every possible variable for every possible counter to 0, and all set variables for all counters to $\emptyset$.

```
start :: State
start = (0, \ _ _ _ -> 0, \ _ _ -> empty)
```

Variable History Inspection

Inspect the complete history of a single variable:

```
inspect :: State -> Name -> [Int]
inspect (c,e,f) x = [(c',e,f) @@ x | c' <- [0..c]]
```

Inspect the complete history of a list of variables:

```
history :: State -> [Name] -> [[Int]]
history (c,e,f) xs =
  [ [(c',e,f) @@ x | x <- xs] | c' <- [0..c]]
```

Variable Dump at a State

Dump a list of variable values at a state:

```
dump :: State -> [Name] -> [Int]
dump s = List.map (\ x -> s@@x)
```

Example:

```
*GCL> dump start ["x1","x2"]
[0,0]
*GCL> history start ["x1","thisOtherVariable"]
[[0,0]]
```

Array Variable History Inspection

```
ainspect :: State -> (Name,[Int]) -> [Int]
ainspect (c,e,f) x =
  [ (c',e,f) @| x | c' <- [0..c]]

ahistory :: State -> [(Name,[Int])] -> [[Int]]
ahistory (c,e,f) xs =
  [ [ (c',e,f) @| x | x <- xs] | c' <- [0..c]]

adump :: State -> [(Name,[Int])] -> [Int]
adump s = List.map (\ x -> s @| x)
```

Set Variable History Inspection

```
sinspect :: State -> Name -> [[Int]]
sinspect (c,e,f) x =
  [toList $ (c',e,f) @@@ x | c' <- [0..c]]

shistory :: State -> [Name] -> [[[Int]]]
shistory (c,e,f) xs =
  [ [toList $ (c',e,f) @@@ x | x <- xs] |
                                c' <- [0..c]]

sdump :: State -> [Name] -> [[Int]]
sdump s = List.map (\ x -> toList $ s@@@x)
```

Timing of Algorithms

Timing is just counting the number of algorithm steps:

```
time :: Action -> Integer
time action = fst' $ action start
   where fst' (x,_,_) = x
```

Skip Action

The skip action is the trivial state transition that changes nothing, so we define it as the identity function on states:

```
skip :: Action
skip = id
```

Abort Action

The abort action is the action that generates an error and stops execution.

```
abort :: Action
abort = error "execution aborted"
```

Sequential Execution (Concatenation)

Imperative "statements" correspond to actions.

Sequential execution corresponds to action composition. Since we want to mention the actions in the order in which they get performed, we define the action separator as a new infix operator:

```
infixl 2 ##

(##) :: Action -> Action -> Action
a1 ## a2 = a2 . a1
```

Variable Assignment: Intensions and Extensions

To understand how to treat variable assignment one has to know the difference between intensions and extensions.

An intension is a function from states to values, an extension is just a value.

Array Variable Assignment, Extensionally

```
aass :: Name -> [Int] -> Int -> Action
aass x is v (c,e,f) = (c',e',f') where
 c' = succ c
 e' t y js
  | t == c' && x == y && js == is = v
  | t == c'                       = e c y js
  | otherwise                     = e t y js
 f' t | t == c'   = f c
 f' t | otherwise = f t
```

The new value v gets assigned to x at is for c'. The new values at c' for variable names and index lists different from (x,is) are inherited from c, and everything else remains the same.

Array Variable Assignment, Intensionally

```
aass' :: Name -> [Int] -> I Int -> Action
aass' x is v = \ s -> aass x is (v s) s
```

Useful for (single index) array assignment:

```
infixl 7 <<<

(<<<) ::  (Name,Int) -> Int -> Action
(x,i) <<< n  = aass x [i] n
```

Multiple Array Variable Assignment, Extensionally

```
maass :: Name -> [([Int],Int)] -> Action
maass x [] = skip
maass x ((is,v):pairs) =
   aass x is v ## maass x pairs
```

Variable Assignment, Extensionally

Values of single variables are implemented as values of array variables at index list `[]`.

In the extensional version of variable assignment, the new value does not depend on state.

```
ass :: Name -> Int -> Action
ass x = aass x []
```

Variable Assignment, Intensionally

The intensional version of assignment assigns an intension.

The value that corresponds to that intension is computed by applying the intension to the current state.

```
ass' :: Name -> I Int -> Action
ass' x v = \ s -> ass x (v s) s
```

Multiple Assignment, Extensionally

```
mass :: [(Name,Int)] -> Action
mass = foldr (\(x,v) action -> ass x v ## action)
             skip
```

Multiple Assignment, Intensionally

```
mass' :: [(Name,I Int)] -> Action
mass' xs s =
  mass (List.map (\ (n,v) -> (n, v s)) xs) s
```

Note that every `I a` argument is evaluated in the same state.

Infix operators for Extensional and Intensional Assignment

```
infixl 7 <=<

(<=<) ::  Name -> Int -> Action
x <=< n  = ass x n

infixl 7 <==

(<==) ::  Name -> I Int -> Action
x <== v  = ass' x v
```

Increment

Increment is defined in terms of intensional assignment, for it depends on state:

```
incr :: Name -> Action
incr x = ass' x (\s -> succ (s@@x))
```

To increment the value of x in state s means to move to a new state s' where the value of x equals the successor of the value of x at s.

Note the constraint `Enum` on the type `a`. The variable has to be of a type for which `succ` is defined.

Set Variable Assignment, Extensionally

In the extensional version of variable assignment, the new value does not depend on state:

```
sass :: Name -> Set Int -> Action
sass x v (c,e,f) = (c',e',f') where
     c' = succ c
     f' t y | x == y && t == c' = v
            | x /= y && t == c' = f c y
            | otherwise         = f t y
     e' t   | t == c'           = e c
     e' t   | otherwise         = e t
```

Set Variable Assignment, Intensionally

The intensional version of assignment assigns an intension.

The value that corresponds to that intension is computed by applying the intension to the current state.

```
sass' :: Name -> SI Int -> Action
sass' x v = \ s -> sass x (v s) s
```

Incrementing Set Variables

Incrementing a set variable is adding an item to its current set value.

```
sincr :: Name -> Int -> Action
sincr x item =
  sass' x (\ s -> Set.insert item (s@@@x))
```

Intensional version:

```
sincr' :: Name -> I Int -> Action
sincr' x f =
  sass' x (\ s -> Set.insert (f s) (s@@@x))
```

Decrementing Set Variables

Decrementing a set variable is deleting an item from its current set value.

```
sdecr :: Name -> Int -> Action
sdecr x item =
  sass' x (\ s -> Set.delete item (s@@@x))
```

Infix Operators for Extensional and Intensional Set Assignment

```
infixl 9 <==<

(<==<) ::  Name -> Set Int -> Action
x <==< set  = sass x set

infixl 9 <===

(<===) ::  Name -> SI Int -> Action
x <=== v  = sass' x v
```

Deterministic Choice, Conditional Execution

Deterministic choice between two actions:

```
if_then_else :: Condition ->
                Action -> Action -> Action
if_then_else c a1 a2 = \ s -> if c s then a1 s
                                     else a2 s
```

Conditional execution of an action:

```
if_then :: Condition -> Action -> Action
if_then c action = \ s -> if c s then action s
                                 else s
```

Select the ‘good’ actions from a list

```
good_actions :: State -> [(Condition,Action)]
                      -> [Action]
good_actions s = List.map snd .
                 List.filter (\ (c,_) -> c s)
```

Non-Deterministic Choice

Dijkstra's non-deterministic choice: pick any good action from a list. At least one action must be good, so we check for that.

```
if_fi :: [(Condition, Action)] -> Action
if_fi xs = \ s ->
  let
    good = good_actions s xs
  in
  if List.null good then abort s
                     else (snd.pick) good s
```

While

If the condition holds then perform the action once and execute the while statement again, otherwise do nothing.

```
while :: Condition -> Action -> Action
while c a =
  \s -> if c s
        then (a ## (while c a)) s
        else s
```

For Each Loop

Nondeterministic version (useful if we want to test that the execution of a foreach loop does not depend on the order of the actions):

```
foreachn :: [Action] -> Action
foreachn []  = skip
foreachn [x] = x
foreachn xs  = x ## foreachn xs' where
     (x,xs') = pickRemove xs
```

Deterministic version:

```
foreach :: [Action] -> Action
foreach = foldr (##) skip
```

Non-Deterministic Repetition

Continue picking good actions from a list of guarded actions and execute them, until no good actions remain.

```
do_od :: [(Condition, Action)] -> Action
do_od xs =
 \s -> let
         good = good_actions s xs
       in
   if List.null good then s
   else ((snd.pick) good ## do_od xs) s
```

Hoare Style Assertions

Hoare style assertions [4] have the form

$$\{\text{Pre}\}\ \text{Command}\ \{\text{Post}\}$$

where Pre and Post are conditions on states.

This Hoare statement is true in state s if truth of Pre in s guarantees truth of Post in any state s' that is a result state of performing Command in state s.

Implementing Hoare Assertions

A Hoare assertion has the following type:

```
assertion :: State -> Condition -> Action
                      -> Condition -> Bool
```

The implementation is a straightforward encoding of the truth conditions:

```
assertion s pre a post = if pre s
                         then post (a s)
                         else True
```

Note that if the precondition does not hold in the initial state, the Hoare assertion is trivially true.

Invariants

An invariant of a command A in a state s is a condition C with the property that if C holds in s then C will also hold in any state that results from execution of A in s.

```
invariant :: Condition -> Action -> Condition
invariant c action s = assertion s c action c
```

Note that an invariant is true in state s if the condition does not hold in s.

Turning a condition into a test

```
test :: Condition -> Action
test c = if_then_else c skip abort
```

Note that the test action does not take any time: when `skip` is performed, the state counter is not incremented.

Invariant Tests

We can use an invariant to produce “self-testing code”, as follows:

```
inv :: Condition -> Action -> Action
inv c action = if_then_else (invariant c action)
                             action
                             abort
```

This wraps the invariant around the action: if the invariant holds for a state and an action, the result of the action on the state is returned, otherwise execution is aborted.

Note that the self-testing code does not take more time than the original code.

Self-Testing Non-Deterministic Repeat

```
do_od_inv :: Condition
             -> [(Condition, Action)]
             -> Action
do_od_inv c xs =
  if_then_else c
  (do_od
     (List.map (\ (d,a) -> (d, a ## test c)) xs))
  (do_od xs)
```

Names for some variables

```
x,y,z :: Name
x = "x"
y = "y"
z = "z"
```

Imperative Algorithm for Squaring a Number

Compute the square of a number in terms of addition.

```
squaring :: Int -> Action
squaring n =
      x <=< 0 ##
      y <=< 0 ##
      z <=< n ##
      (while (\s -> s@@y < s@@z)
          (
           x <== (\s -> s@@x+s@@y+s@@y+1)
           ## incr y))
```

Constructing An Invariant

Isolate the body of the while loop:

```
whileBody :: Action
whileBody =
  ass' x (\s -> s@@x+s@@y+s@@y+1) ## incr y
```

Construct a suitable internal state for checking an invariant of the while body:

```
mkState :: Int -> Int -> State
mkState m n = foreach
   [ x <=< m, y <=< n ] start
```

From Invariants to QuickCheck Properties

Express the invariant as a property to be checked with QuickCheck [1].

```
prop_invar :: (Int,Int) -> Bool
prop_invar (m,n) =
  invariant (\ s -> s@@x == (s@@y)^2)
            whileBody
            (mkState m n)
```

This succeeds.

Some More QuickCheck Properties

Here is another property of `squaring` for checking with QuickCheck.

```
prop_sq n = squaring n start @@ x == n^2
```

This gives counterexamples. Here is the remedy:

```
prop_sq1 n = if n >= 0
             then squaring n start @@ x == n^2
             else True
```

Setting a Threshold

```
threshold :: Int
threshold = 1000
```

Testing `squaring` for arbitrary `Int` values takes far too long.

Our algorithm execution code is not very fast (nor is it supposed to be fast).

QuickCheck has a remedy for that:

```
prop_sq2 n = n < threshold ==>
             if n >= 0
             then squaring n start @@ x == n^2
             else True
```

Still Nicer

Test with the appropriate assertion:

```
prop_sq3 n = n < threshold ==>
             assertion
             start
             (\ _ -> n >= 0)
             (squaring n)
             (\ s -> s@@x == (s@@y)^2)
```

‘Self-testing’ Version of Squaring

Use test injection to produce a self-testing version of the algorithm:

```
squaring2 :: Int -> Action
squaring2 n =
  x <=< 0 ##
  y <=< 0 ##
  z <=< n ##
  while (\s -> s@@y < s@@z)
    (inv (\ s -> s@@x == (s@@y)^2)
     (x <== (\s -> s@@x+s@@y+s@@y+1) ## incr y)
    )
```

Yet Another Way

```
squaring3 :: Int -> Action
squaring3 n =
  x <=< 0 ##
  y <=< 0 ##
  z <=< n ##
  while (\s -> s@@y < s@@z)
    (
      test (\ s -> s@@x == (s@@y)^2)
      ##
      x <== (\s -> s@@x + s@@y + s@@y + 1)
      ##
      incr y
      ##
      test (\ s -> s@@x == (s@@y)^2)
    )
```

Euclid's gcd algorithm

```
euclid :: Int -> Int -> Action
euclid m n =
  x <=< m  ##
  y <=< n  ##
  while (\s -> not (s@@x == s@@y))
           (if_then_else (\ s -> s@@x < s@@y)
               (y <== \s -> s@@y - s@@x)
               (if_then (\ s -> s@@x > s@@y)
                       (x <== \s -> s@@x-s@@y)))
```

Dijkstra's version

This looks more elegant in Dijkstra's guarded command style:

```
euclid2 :: Int -> Int -> Action
euclid2 m n =
  x <=< m  ##
  y <=< n  ##
  do_od
   [
    (\s -> s@@x < s@@y, y <== \s -> s@@y-s@@x),
    (\s -> s@@y < s@@x, x <== \s -> s@@x-s@@y)
   ]
```

Testing With Hoare Assertions

Checking `euclid` with Hoare assertions in QuickCheck:

```
prop_euclid :: (Int,Int) -> Property
prop_euclid (m,n) =
  m < threshold && n < threshold ==>
  assertion start
            (\_ -> m > 0 && n > 0)
            (euclid m n)
            (\ s -> s@@x == gcd m n)
```

And similarly for `euclid2`.

```
prop_euclid2 :: (Int,Int) -> Property
prop_euclid2 (m,n) =
  m < threshold && n < threshold ==>
  assertion start
            (\_ -> m > 0 && n > 0)
            (euclid2 m n)
            (\ s -> s@@x == gcd m n)
```

A Nicer Way: Self-Testing Version of Euclid

```
euclid3 :: Int -> Int -> Action
euclid3 m n =
  x <=< m  ##
  y <=< n  ##
  do_od_inv
   (\s -> gcd m n == gcd (s@@x) (s@@y))
    [
     (\s -> s@@x < s@@y, y <== \s -> s@@y - s@@x),
     (\s -> s@@y < s@@x, x <== \s -> s@@x - s@@y)
    ]
```

The Extended Euclidean Algorithm

```
extEuclid :: Int -> Int -> Action
extEuclid m n = let
    (a, b, u, v, q, r) = ("a","b","u","v","q","r")
   in
   foreach [a<=<m,b<=<n,x<=<1,y<=<0,u<=<0,v<=<1] ##
   while (\s -> s@@b /= 0)
     (foreach
        [ q <==  \s -> s@@a ‘div‘ s@@b,
          r <==  \s -> s@@a ‘mod‘ s@@b ]          ##
      mass’ [(a,(@@b)),
             (b,(@@r)),
             (x,(@@u)),
             (y,(@@v)),
             (u,\s -> s@@x - (s@@q * s@@u)),
             (v,\s -> s@@y - (s@@q * s@@v))     ]
      )
```

Checking the Extended GCD Algorithm

Bézout's identity is the equality $xM + yN = \gcd(M, N)$.

A QuickCheck property for it:

```
prop_Bezout :: (Int,Int) -> Bool
prop_Bezout (m,n) =
  assertion start
            (\ _ -> m > 0 && n > 0)
            (extEuclid m n)
            (\ s -> (s@@x)*m + (s@@y)*n == gcd m n)
```

Purely Functional Algorithms

Haskell is already a specification language for purely function algorithms, so here we do not need the extensions.

The following is a literal version of an alternative algorithm for gcd.

```
ext_gcd :: (Int,Int) -> (Int,Int)
ext_gcd (a,b) =
  if b == 0
  then (1,0)
  else
     let
       (q,r) = quotRem a b
       (s,t) = ext_gcd (b,r)
     in (t, s - q*t)
```

A check for Bézout's identity again:

```
prop_Bezout1 :: (Int,Int) -> Bool
prop_Bezout1 (m,n) =
   if m > 0 && n > 0
   then x*m + y*n == gcd m n
   else True
     where (x,y) = ext_gcd (m,n)
```

Problem of Finding a Minimum Spanning Tree of a Graph

- A weighted undirected graph is a graph with weights assigned to the edges. Think of the weight as an indication of distance.
- Let G be a weighted, undirected (i.e., symmetric) and connected graph. Assume there are no self-loops. (Or, if there are self-loops, make sure their weight is set to 0.)
- A minimum spanning tree for weighted graph G is a spanning tree of G whose edges sum to minimum weight.
- Caution: minimum spanning trees are not unique.
- Applications: finding the least amount of wire necessary to connect a group of workstations (or homes, or cities, or ...).

Prim's Minimum Spanning Tree Algorithm

Finds a minimum spanning tree for an arbitrary weighted symmetric and connected graph. See [6], [7, 4.7].

- Select an arbitrary graph node *r* to start the tree from.
- While there are still nodes not in the tree
  - Select an edge of minimum weight between a tree and non-tree node.
  - Add the selected edge and vertex to the tree.

It is not at first sight obvious that Prim's algoritm always results in a minimum spanning tree, but this fact can be checked by means of Hoare assertions, which can be tested.

Datatype for Weighted Graphs

```
type Vertex = Int
type Edge = (Vertex,Vertex,Int)
type Graph = ([Vertex],[Edge])
```

If (x, y, w) is an edge, then the edge is from vertex x to vertex y, and its weight is w.

Creation of Proper Symmetric Edge Lists

Make a list of edges into a proper symmetric graph, while also removing self loops and edges with non-positive weights.

```
mkproper :: [Edge] -> [Edge]
mkproper xs = let
  ys = List.filter
    (\ (x,y,w) -> x /= y && w > 0) xs
  zs = nubBy (\ (x,y,_) (x',y',_) ->
    (x,y) == (x',y') || (x,y) == (y',x')) ys
  in foldr
      (\ (x,y,w) us -> ((x,y,w):(y,x,w):us))
      [] zs
```

Checking for Connectedness

Connected components of a graph are given by a least fixpoint computation:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

Tracing Edges to Extend a Vertex Set

Set of all vertices that are connected by an edge to some vertex in a given set:

```
connectedC :: [Edge] -> Set Vertex -> Set Vertex
connectedC es set =
  fromList [ v | x <- toList set,
                 (u,v,w) <- es, x == u ]
```

Connectedness

The whole graph is connected if the least fixpoint computation yields the full set of vertices:

```
connected :: [Edge] -> Set Vertex -> Bool
connected es set = set == set'
  where
    x = takeElem set
    set' = lfp
            (\ s -> Set.union s (connectedC es s))
            (singleton x)
```

Note: this presupposes symmetry of the list of edges.

Finding Index of Edge with Minimum Weight

Finding the vertex y and index i with the property that i is the index of the (x, w, y) with the least w such that $x \in I$, $y \in O$.

```
minVE :: Graph -> Set Vertex -> Set Vertex
                             -> (Vertex,Int)
minVE graph ins outs =
 let
   ies = zip (snd graph) [0..]
   new = [ ((x,y,w),i) |((x,y,w),i) <- ies,
                         member x ins,
                         member y outs ]
   new'= sortBy (\((_,_,w),_) ((_,_,w'),_) ->
                 compare w w') new
 in (\((_,y,_),i) -> (y,i)) $ head new'
```

Prim’s Algorithm

```
prim :: Graph -> Action
prim (vs,es) = let
    vset  = fromList vs
    root  = takeElem vset
    (iset,oset,tree) = ("iset", "oset","tree")
  in
   iset <==< singleton root                ##
   oset <==< Set.delete root vset          ##
   tree <==< empty                         ##
    while (\s -> s@@@oset /= empty)
      ( \ s ->
        let
         (v,i) = minVE (vs,es)(s@@@iset)(s@@@oset)
        in
          (sdecr oset v ## sincr iset v ##
           sincr tree i)   s)
```

Testing this:

```
graph0 :: Graph
graph0 =
  ([1,2,3],mkproper [(1,2,3),(2,3,3),(1,3,7)])
```

```
*GCL> prim graph0 start @@@ "tree"
fromList [0,2]
*GCL> prim graph0 start @@@ "tree"
fromList [1,3]
```

Note that we do not always get the same outcome: the algorithm is non-deterministic. To interpret the output we need to see the edge list:

```
*GCL> mkproper [(1,2,3),(2,3,5),(1,3,7)]
[(1,2,3),(2,1,3),(2,3,5),(3,2,5),(1,3,7),(3,1,7)]
```

Dijkstra's Shortest Path Algorithm

See [2], or [7, 4.7]. Assume a nondirected graph (the edge relation is symmetric). Assume that each edge has non-negative edge weight.

- Principle: Given that we have shortest paths between s and $v_1, \ldots, v_k$, and there are nodes x such that the shortest path from s to x is not known. Then there is a node x such that the shortest path from s to x goes from s to v_i to x, for some i with $1 \leq i \leq k$.
- More specifically, it is the node x with minimal $d(s, v_i) + w(v_i, x)$, where d gives the distance along a path, and w the weight of an edge.

Dijkstra's Shortest Path Algorithm (ctd)

Shortest path from x to y.

- Let nodes be numbered $v_1, \ldots, v_n$. Let d be an array of distances from x.
- Let $K = \{x\}$. Let last $= x$. Let $d[i] = \infty$ for all $i \in \{0, \ldots n-1\}$.
- For all edges (s, v_i), let $d[i] = w(s, v_i)$.
- While last $\neq y$ do
  - select v_{next}, the unknown node minimizing $d[\text{next}]$.
  - for each edge (v_{next}, v_j),
    set $d[j]$ equal to $\min(d[j], d[\text{next}] + w(v_{\text{next}}, v_j))$.
  - set last $= v_{\text{next}}$.
  - set $K = K \cup \{v_{\text{next}}\}$.

Auxiliary Functions

```
infty = maxBound :: Int
```

For given x, find all pairs (y, w) such that (x, y, w) is an edge.

```
weights :: [Edge] -> Vertex -> [(Vertex,Int)]
weights es u = [ (y,w) | (x,y,w) <- es, u == x]
```

Closest Unknown Vertex

Find the closest unknown vertex, measured by distance function *d*.

```
closest :: [Vertex] -> Set Vertex ->
           (Vertex -> Int) -> Vertex
closest univ known d = let
  pairs = [(x,d x) | x <- univ, notMember x known]
  spairs =
     sortBy (\ (_,d1) (_,d2) -> compare d1 d2)
       pairs
 in fst $ head spairs
```

Distance Count Adjustment (1)

Adjust the distance count for a given edge:

```
adj :: Edge -> (Vertex -> Int) -> (Vertex,Int)
adj (x,y,w) d = (y, f (d y) (d x) w)
  where f dy dx v | dx == infty = dy
                  | otherwise   = min dy (dx+v)
```

A subtlety here: adding a positive weight to ∞ gives a negative number, so we have to check for equality to ∞ before we add weight to (dx).

Distance Count Adjustment (2)

Adjust the distance count for all edges starting from `last`:

```
adjust :: [Edge] -> Vertex -> (Vertex -> Int)
                               -> [(Vertex,Int)]
adjust es last d =
 [ adj (x,y,w) d | (x,y,w) <- es, x == last ]
```

Dijkstra's Shortest Path Algorithm

```
shortestPath :: Graph -> (Vertex,Vertex)
                           -> Action
shortestPath (vs,es) (x,y) =
 let
   (known,dist,last,next) = ("k","dist","l","n")
 in
  (
    foreach [ aass dist [v] infty | v <- vs ]    ##
    foreach [ aass dist [y] w |
                            (y,w) <- weights es x ]  ##
    known <==< singleton x                       ##
    last  <=<  x                                 ##
```

```
    while (\s -> s@@last /= y)
     (\s -> let
         kn     = s@@@known
         dst x = s@|(dist,[x])
         nxt    = closest vs kn dst
         lst    = s@@last
       in
       (
         next <=< nxt                                         ##
         foreach
           [ (dist,v) <<< d |
                 (v,d) <- adjust es nxt dst ]                 ##
         last <== (@@next)                                    ##
         sincr' known (@@next)) s)
   )
```

Testing It Out

```
graph :: Graph
graph = ([0..4],mkproper
  [(0,3,1),(1,2,3),(1,3,2),(2,3,4),(3,4,5)])
```

```
*GCL> shortestPath graph (1,4) start @|("dist",[4])
7
*GCL> shortestPath graph (4,1) start @|("dist",[1])
7
```

Automated Time Complexity Analysis

This is work in progress.

```
*GCL> [ time (squaring n) 0  | n <- [0..15] ]
[3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33]
```

Further Work

- generate pseudocode in a variety of flavours from the formal algorithm specifications,
- generate executable (C, Java, . . .) code from the formal algorithm specifications,
- develop automated time complexity analysis . . .
- implement monadic algorithm debugger . . .
- use all this in master course in purely functional algorithm specification and analysis.

Conclusion

Purely functional programming is a suitable tool for specification of algorithms written in imperative style.

The method is to implement the semantics of an imperative algorithm as a function from states to states, and to define the appropriate algorithm construction operations.

Hoare style assertions for such algorithms take the shape of executable tests, so the algorithms can be tested and debugged easily.

Also, the time complexity of the algorithms is open for inspection.

Haskell is excellent for writing testable properties of algorithms, and Haskell's automated test tooling can be used off-the-shelf.

References

[1] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Proc. Of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000.

[2] E.W. Dijkstra. A note on two problems in connection with graphs. Numerische Mathematik, 1:269–271, 1959.

[3] E.W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. Communications of the ACM, 18:453–457, 1975.

[4] C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):567–580, 583, 1969.

[5] Leslie Lamport. The PlusCal algorithm language. In Martin Leucker and Carroll Morgan, editors, Theoretical Aspects of Computing – ICTAC 2009, number 5684 in Lecture Notes in Computer Science, pages 36–60. Springer, 2009.

[6] R.C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389–1401, 1957.

[7] Steven S. Skiena. The Algorithm Design Manual. Springer Verlag, New York, 1998. Second Printing.