Work Efficient Higher-Order Vectorisation (Appendix)

Ben Lippmeier† Manuel M. T. Chakravarty† Gabriele Keller† Roman Leshchinskiy†

Simon Peyton Jones‡

†Computer Science and Engineering
University of New South Wales, Australia
{benl,chak,keller,rl}@cse.unsw.edu.au

‡Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

1. Technical Report

This appendix will be appended to the main paper to become the technical report cited in the paper.

2. Example Code

A sequential reference implementation of our new array representation is available from:

```
http://www.cse.unsw.edu.au
        /~benl/papers/replicate/dph-reference-array.tgz
```

This package also contains the example derivations as Haskell modules and works with at least GHC 7.0.4 and GHC 7.4.1.

The production implementation is part of DPH 0.6.1.2 which is available on Hackage. This library needs GHC 7.4.1 to run. The `dph-lifted-vseg` package implements the lifted combinators, while the `dph-prim-seq` and `dph-prim-par` packages implement the flat array operators.

The benchmark programs presented in the paper are part of the `dph-examples` package, also on Hackage.

```
-- Closures and closure arrays ------------------------
data (a :-> b)
 = forall env. PA env
 => Clo  (env -> a -> b)
         (Int -> PData env -> PData a -> PData b)
         env

data instance PData{s} (a :-> b)
 =  forall env. PA env
 => AClo  (env -> a -> b)
          (Int -> PData env -> PData a -> PData b)
          (PData{s} env)

-- Closure constructors -------------------------------
closure1 :: (a -> b)
         -> (Int -> PA a -> PA b)
         -> (a :-> b)
closure1 fv fl
 = let  fl' n pdata
         = case fl n (PArray n pdata) of
                PA _ pdata' -> pdata'
   in   Clo  (\_env -> fv)
             (\n _env -> fl' n) ()

closure2 -> (a -> b -> c)
         -> (Int -> PA a -> PA b -> PA c)
         -> (a :-> b :-> c)
closure2 fv fl
 = let  fl' n pdata1 pdata2
         = case fl n (PA n pdata1) (PA n pdata2) of
                PA _ pdata' -> pdata'
        fv_1 _ xa   = Clo  fv fl' xa
        fl_1 _ _ xs = AClo fv fl' xs
   in   Clo fv_1 fl_1 ()

-- Closure and lifted closure application -------------
($:) :: (a :-> b) -> a -> b
($:) (Clo fv _fl env) x  = fv env x

($:^) :: PA (a :-> b) -> PA a -> PA b
PA n (AClo _fv fl envs) $:^ PA _ as
        = PA n (fl n envs as)

-- Closure converted combinators ----------------------
indexPP :: PA a => PA a :-> Int :-> a
indexPP = closure2 PA.index PA.index_l

mapPP  :: (a :-> b) :-> PA a :-> PA b
mapPP   = closure2 mapPP_v mapPP_l
 where mapPP_v :: (a :-> b) -> PA a -> PA b
       mapPP_v f as
        =   replicatePA (lengthPA as) f $:^ as
       mapPP_l :: PA (a :-> b) -> PA (PA a) -> PA (PA b)
       mapPP_l fs ass
        =   unconcatPA ass
        $   replicatesPA (takeLengths ass) fs
        $:^ concatPA ass

zipWithPP :: (a :-> b :-> c)
          :-> PArray a  :-> PArray b :-> PArray c
zipWithPP   = closure3 zipWithPP_v zipWithPP_l
 where zipWithPP_v f xs ys
        =   replicatePA (lengthPA xs) f $:^ xs $:^ ys
       zipWithPP_l _ fs ass bss
        =   unconcatPA ass
        $   replicatesPA (takeLengths ass) fs
        $:^ concatPA ass
        $:^ concatPA bss
```

Figure 1. Closure Converted Lifted Combinators.

```
-- Vectorising Types ------------------------------------
Vt[T] :: Type -> Type
Vt[T1 -> T2] = Vt[T1] :-> Vt[T2] (functions)
Vt[ [:T:] ]  = Lt[T]             (parallel arrays)
Vt[ Int ]    = Int               (primitive scalar types)
Lt[T]        = PA Vt[T]


-- Vectorising Terms ------------------------------------
V[E]  :: Term -> Term
V[k]     = k                   (literals)
V[f]     = f_PP                (f is bound at top level)
V[x]     = x                   (x is locally bound)
V[E1 E2] = V[E1] $: V[E2]      (application)

V[let f x1 x2 .. = E1 in E2]
 = let fv   x1 x2 .. = V[E1]
       fl n x1 x2 .. = L[E1]n
       f_PP          = closure_N fv fl
   in  V[E2]

V[if E1 then E2 else E3]
 = if V[E1] then V[E2] else V[E3]


-- Lifting Terms ----------------------------------------
L[E]n  :: Term -> Term -> Term
L[k]n    = replicatePA n k     (literals)
L[f]n    = replicatePA n f_PP  (f is bound at top level)
L[x]n    = x                   (x is locally bound)
L[E1 E2] = L[E1]n $:^ L[E2]n   (application)

L[let f x1 x2 .. = E1 in E2]n
 = let fv   x1 x2 .. = V[E1]
       fl m x1 x2 .. = L[E1]m
       f_PP          = closure_N fv fl
   in  L[E2]n

L[if E1 then E2 else E3]n
 = let flags = L[E1]n
   in  combine flags (L[E2'] (countTrue  flags))
                     (L[E3'] (countFalse flags))
        with E2' = [{packPA fvs_i flags True  / fvs_i}]E2
             E3' = [{packPA fvs_i flags False / fvs_i}]E3
```

Figure 2. The Vectorisation Transform

3. Vectorisation of the retrieve function

The following is a derivation of the vectorised version of the `retrieve` function discussed in §2.

```
retrieve :: [:[:Char:]:] -> [:[:Int:]:] -> [:[:Char:]:]
retrieve xss iss
  = zipWithP mapP (mapP indexP xss) iss
```

We first apply the vectorisation transform from Figure 2. This replaces application of library functions to their closure converted (`*PP`) versions. The definitions of these functions are in Figure 1.

```
retrieve_v :: PA (PA a) -> PA (PA Int) -> PA (PA a))
retrieve_v xss iss
 = zipWithPP $: mapPP $: (mapPP $: indexPP $: xss) $: iss
```

We proceed by inlining the definitions of the library functions and simplify where appropriate. By doing this we will see how `replicates` and `concat` are introduced into the program. We start by splitting out the partial application into its own binding to help the presentation.

```
retrieve_v xss iss
 = let fs     = mapPP $: indexPP $: xss
   in  zipWithPP $: mapPP $: fs  $: iss
```

Inlining `zipWithPP` and the first instance of `mapPP` reveals that the closures for the worker functions are replicated. Inlining also introduces the lifted application operator (`$:^`). The definition of `mapPP` is given in Figure 1, and `zipWithPP` is a simple extension.

```
retrieve_v xss iss
 = let fs = (replicate (length xss) indexPP) $:^ xss
   in  (replicate (length iss) mapPP) $:^ fs $:^ iss
```

We now inline the lifted application operator (`$:^`). As `indexPP` is partially applied, we end up with an explicit closure which captures the `xss` array in its environment. In contrast, `mapPP` has been fully applied, so the lifted application reduces to a direct application of the lifted map function `mapPP_l`.

```
retrieve_v xss iss
 = let fs = Clo index index_l xss
   in  mapPP_l fs iss
```

Inlining `mapPP_l` reveals that segmented replicate is being applied to the closure representing the partial application of `indexP` in the original program. Note that we are now using `replicatesPR`. The `*PR` suffix indicates that the function works on the internal `PData` type rather than the `PA` wrapper.

```
retrieve_v xss iss
 =   unconcat iss
 $   (let ns   = lengths $ takeSegd iss
          n    = sum ns
      in  PA n (replicatesPR ns
                        (Clo index index_l xss)))
 $:^ concat iss
```

We now inline the `replicatesPR` instance for closures. Performing segmented replicate on a closure produces an array closure where the environment has been replicated.

```
retrieve_v xss@(PA _ xss') iss
 =   unconcat iss
 $   (let ns  = lengths $ takeSegd iss
          n   = sum ns
      in  PArray n (AClo index index_l
                        (replicatesPR ns xss')))
 $:^ concat iss
```

Finally, we inline the remaining lifted application operator. This reveals that lifted indexing is being applied to our replicated tables array (`xss`). The vectorised function retrieves one element from each of the copies, then unconcatenates the result to produce the nesting structure of the original indices array (`iss`).

```
retrieve_v xss@(PArray _ xss') iss@(PArray _ iss')
   = unconcat iss
   $ let ns  = lengths $ takeSegd iss
         n   = sum ns
     in  PArray n (indexlPR n (replicatesPR ns xss')
                              (concatPR iss'))
```

We now consider what complexity bounds must be placed on the array operators so that the vectorised version of `retrieve` has the same complexity as the original. The work complexity of the original is $O(length\ (concat\ \texttt{iss}))$. For the vectorised version to retain this complexity the operators `indexlPR`, `replicatesPR` and `concatPR` must all be linear in the length of their results. Since `retrieve` is polymorphic in the element type `a`, the array operators must have this complexity for possible element types. This includes arrays of arbitrary nesting depth.

4. Vectorisation of the retsum function

The `retsum` function indexes several shared arrays, and adds the retrieved value to the sum of the array it came from. This has a similar structure to `retrieve` from the previous section.

```
retsum :: [:[:Int:]:] -> [:[:Int:]:] -> [:[:Int:]:]
retsum xss iss
 = zipWithP mapP
           (mapP (\xs i. indexP xs i + sumP xs) xss) iss
```

Applying the vectorisation transform yields:

```
retsum_v xss iss
 = let fv ys j     = index ys j + sum ys
       fl c yss js = add_l c (index_l c yss js)
                             (sum_l   c yss)
       fPP         = closure2 fv fl
   in  zipWithPP $: mapPP $: (mapPP $: fPP $: xss) $: iss
```

Shift partial application into own binding and inline `zipWithPP`

```
retsum_v xss iss
 = let c            = length iss
       fv ys j      = index ys j P.+ sum ys
       fl c' yss js = add_l c' (index_l c' yss js)
                               (sum_l   c' yss)
       fPP     = closure2 fv fl
       gs      = mapPP $: fPP $: xss
   in  replicate c mapPP $:^ gs $:^ iss
```

Inline `closure2` and `replicates` instances.

```
retsum_v _xss@(PA _ xss') iss
 = let c            = length iss
       fv ys j      = index ys j P.+ sum ys
       fl c' yss js = add_l c' (index_l c' yss js)
                               (sum_l   c' yss)

       fl' n pdata1 pdata2
        = case fl n (PA n pdata1) (PA n pdata2) of
           PA _ pdata' -> pdata'

       fl_1 _ _ xs = AClo fv fl' xs
       gs          = PA c (fl_1 c (replicatePR c ()) xss')
   in  replicate c mapPP $:^ gs $:^ iss
```

Inline `fl_1`, `mapPP`, and `replicate` on closures.

```
retsum_v _xss@(PA _ xss') iss
 = let fv ys j = index ys j P.+ sum ys
       fl c' yss js = add_l c' (index_l c' yss js)
                               (sum_l   c' yss)

       fl' n pdata1 pdata2
        = case fl n (PA n pdata1) (PA n pdata2) of
           PA _ pdata' -> pdata'

   in  unconcat iss
        $ (let ns = lengths iss
               n  = sum ns
           in  PA n (AClo fv fl' (replicatesPR ns xss')))
        $:^ concat iss
```

Inline lifted applications.

```
retsum_v xss iss
 = let fl c' yss js = add_l c' (index_l c' yss js)
                               (sum_l   c' yss)
   in  unconcat iss
        $ (let ns = lengths iss
               n  = sum ns
           in  fl n (replicates ns xss) (concat iss))
```

Inline `fl` and float bindings.

```
retsum_v xss iss
 = let  ns      = lengths iss
        n       = sum ns
        yss'    = replicates ns xss
   in   unconcat iss
         $ add_l n (index_l n yss' (concat iss))
                   (sum_l   n yss')
```

5. Vectorisation of the furthest function

The `furthest` function takes an array of points and computes the maximum distance between any pair.

```
furthest :: [:(Float, Float):] -> Float
furthest ps = maxP (mapP (\p. maxP (mapP (dist p) ps)) ps)

dist :: (Float, Float) -> (Float, Float) -> Float
```

Applying the vectorisation transform yields:

```
furthest_v :: PA Int -> Int
furthest_v xs
 = let  fv :: Int -> Int
        fv       = unused

        fl :: Int -> PA Int -> PA Int
        fl c ys  =    replicate c maxPP
                 $:^ (replicate c mapPP
                        $:^ (replicate c distPP $:^ ys)
                        $:^  replicate c xs)

        fPP      :: Int :-> Int
        fPP      = closure1 fv fl

  in    maxPP $: (mapPP $: fPP $: xs)
```

Inline `maxPP`, `fPP` and last occurrence of `mapPP`.

```
furthest_v xs
 = let fl c ys  = max_l c
                $ replicate c mapPP
                    $:^ (replicate c distPP $:^ ys)
                    $:^ replicate c xs
 in    max (fl (length xs) xs)
```

Inline inner `mapPP`.

```
furthest_v xs
 = let fl c ys
        = let xss'     = replicate c xs
          in  max_l c
               $   unconcat xss'
               $   replicates (lengths xss')
                    ((replicate c distPP) $:^ ys)
               $:^ concat xss'
   in  max (fl (length xs) xs)
```

Float bindings.

```
furthest_v xs
 = max (let c    = length xs
            xss' = replicate c xs
        in  max_l c
             $   unconcat xss'
             $   replicates (lengths xss')
                  ((replicate c distPP) $:^ xs)
             $:^ concat xss')
```

Inline distPP closure.

```
furthest_v xs
 = let c       = length xs
       xss'    = replicate c xs
       ns      = lengths xss'

       fl' n pdata1 pdata2
        = case dist_l n (PA n pdata1) (PA n pdata2) of
           PA _ pdata' -> pdata'

       fv_1 _ xa    = Clo  dist fl' xa
       fl_1 _ _ xs' = AClo dist fl' xs'
       clo          = Clo fv_1 fl_1 ()
   in  max $   max_l c
           $   unconcat xss'
           $   replicates ns ((replicate c clo) $:^ xs)
           $:^ concat xss'
```

Inline clo

```
furthest_v xs@(PA _ xs')
 = let c       = length xs
       xss'    = replicate c xs
       ns      = lengths xss'

       fl' n pdata1 pdata2
        = case dist_l n (PA n pdata1) (PA n pdata2) of
           PA _ pdata' -> pdata'

   in  max $   max_l c
           $   unconcat xss'
           $   replicates ns (PA c (AClo dist fl' xs'))
           $:^ concat xss'
```

Inline replicates and final lifted application.

```
furthest_v xs@(PAy _ xs')
 = let c       = length xs
       xss'    = replicate c xs
       ns      = lengths xss'

       fl' n pdata1 pdata2
        = case dist_l n (PA n pdata1) (PA n pdata2) of
           PA _ pdata' -> pdata'

   in  max $ max_l c
           $ unconcat xss'
           $ (case concat xss' of
               PA _ xssd
                -> PA  (sum ns)
                 $ fl' (sum ns) (replicatesPR ns xs') xssd)
```

Inline fl' and simplify.

```
furthest_v xs
 = let  c    = length xs
        xss' = replicate c xs
        ns   = lengths xss'
   in   max  $ max_l c
             $ unconcat xss'
             $ dist_l (U.sum ns)
                     (replicates ns xs) (concat xss')
```

Note that if c is the length of xs all $O(\mathtt{c}^2)$ distances will be computed by dist_l before max and max_l determine the greatest one. When run sequentially, the source function would use space linear in the length of xs, but the vectorised version uses space quadratic in the length of xs. This exposes the maximal amount of parallelism, at the cost of increased space complexity to hold the intermediate values.

6. Segment Descriptor Culling Functions

```
-- | Drop physical segments in a SSegd that are unrechable
--   from the segmap, and rewrite the segmap to match.
cullOnSegmap :: Vector Int -> SSegd -> (Vector Int, SSegd)
cullOnSegmap segmap (SSegd sources starts (Segd lengths _))
 = (segmap', ssegd')
 where
    (used_flags, used_map)
     = makeCullMap (length sources) segmap

    -- Use the used_map to rewrite the segmap to point to
    -- the corresponding psegs in the result.
    --  Example:   segmap:  [0 1 1 3 5 5 6 6]
    --           used_map:  [0 1 -1 2 -1 3 4]
    --            segmap':  [0 1 1 2 3 3 4 4]
    segmap'   = map (used_map !) segmap

    -- Drop unreachable psegs entries from the SSegd.
    starts'   = pack starts  used_flags
    sources'  = pack sources used_flags
    lengths'  = pack lengths used_flags

    ssegd'    = SSegd sources' starts'
              $ segdOfLengths lengths'


-- | Drop data chunks in a PDatas that are unreachable
--   from the SSegd, and rewrite the SSegd to match.
cullOnSSegd :: PR a => SSegd -> PDatas a -> (SSegd, PDatas a)
cullOnSSegd (SSegd sources starts segd) pdatas
 = (ssegd', pdatas')
 where
    (used_flags, used_map)
     = makeCullMap (lengthdPR pdatas) sources

    -- Rebuild the SSegd.
    sources' = map (used_map !) sources
    ssegd'   = SSegd sources' starts segd

    -- Drop unreachable chunks from the PDatas.
    pdatas'  = packdPR pdatas used_flags


makeCullMap:: Int -> Vector Int ->(Vector Bool, Vector Int)
makeCullMap total used
 = (flags, used_map)
 where
    -- Make an array of flags signalling whether each
    -- element is used or not.
    -- Example: used:  [0 1 1 3 5 5 6 6]
    --       => flags: [T T F T F T T]
    flags
     = backpermuteDft total (const False)
     $ zip used
          (replicate (length used) True)

    -- Make a set of used indices.
    --  Example: flags:    [T T F T F T T]
    --       =>  uset_set: [0 1 3 5 6]
    used_set
     = pack (enumFromN 0 (length flags)) flags

    -- Make am array that maps used elements in the source
    -- array onto elements in the result array.
    -- If a particular element isn't used this maps to -1.
    -- Example: used_set:  [0 1 3 5 6]
    --          used_map:  [0 1 -1 2 -1 3 4]
    used_map
     = backpermuteDft total (const (-1 :: Int))
     $ zip used_set
          (enumFromN 0 (length used_set))
```

7. Virtual Shared Indexing

The following `indexvsPR` function implements virtual shared indexing for nested arrays and is described in §5.1 of the main paper.

```
instance PR a => PR (PA a) where
 indexvsPR (PNesteds pdatas) vsegd1 srcixs
  = PNested vsegd' pdatas'
  where
    -- O(length segixs)
    (segLengths, segStarts, segBlocks)
     = unzip3
     $ map (\(ix1, ix2) ->
        let -- Index into the outer array.
            ssegd1  = ssegd   vsegd1
            psegid1 = segmap  vsegd1 ! ix1
            source1 = sources ssegd1 ! psegid1
            start1  = starts  ssegd1 ! psegid1

            -- Index into the inner arrays.
            arr2    = pdatas  ! source1
            vsegd2  = vsegd   arr2
            ssegd2  = ssegd   vsegd2
            segd2   = segd    ssegd2
            psegid2 = segmap  vsegd2 ! (start1 + ix2)
            source2 = sources ssegd2 ! psegid2
            start2  = starts  ssegd2 ! psegid2
            length2 = lengths segd2  ! psegid2
            block2  = pdata   arr2 `indexdPR` source2
        in  (length2, start2, block2))
     $ srcixs

    -- O(length segixs)
    vsegd'  = promoteSSegd
            $ SSegd (enumFromN 0 (length srcixs))
                    segStarts
            $ segdOfLengths segLengths

    -- O(length flats) = O(length segixs)
    pdatas' = concatdPR
            $ map singletondPR segBlocks
```

8. Virtual Shared Extraction

The following `extractvsPR` function implements virtual shared extraction for nested arrays and is described in §5.1 of the main paper.

```
instance PR a => PR (PA a) where
 extractvsPR (PNesteds pdatas) vsegd1
  = PNested vsegd' pdatas_culled
  where
    ssegd1      = demoteVSegd vsegd1
    segLengths  = lengths $ segd ssegd1
    segSources  = sources ssegd1

    -- Get the array id for each segment in the result.
    src_sources = replicates segLengths segSources

    -- Gather up the segmaps from each source array.
    segmaps    = PInts $ map  (segmap . vsegd)  pdatas
    sourcess_v = map (sources . ssegd . vsegd)  pdatas
    startss_v  = map (starts  . ssegd . vsegd)  pdatas
    lengthss_v = map (lengths.segd.ssegd.vsegd) pdatas

    -- Get the psegid to use for each segment in the
    -- result, relative to the source arrays.
    PInt src_psegids = extractvsPR segmaps vsegd1

    -- Because all the flat arrays go into the result,
    -- we need to adjust the source ids from the
    -- original arrays.
    psrcoffset = prescanl (+) 0
               $ map (lengthdPR . pnestedPData) pdatas

    -- Get the block id for each segment in the result.
    dst_sources
     = zipWith (\src pseg -> (sourcess_v ! src) ! pseg
                         +   psrcoffset ! src)
               src_sources src_psegids

    -- Get the start index for each segment in its block.
    dst_starts
     = zipWith (\src pseg -> (startss_v ! src) ! pseg)
               src_sources src_psegids

    -- Get the length of each segment in the result.
    dst_lengths
     = zipWith (\src pseg -> (lengthss_v ! src) ! pseg)
               src_sources src_psegids

    -- Build the SSegd for the result.
    -- This references all data blocks in the source.
    ssegd_all   = SSegd dst_sources dst_starts
                $ segdOfLengths dst_lengths

    -- Collect up all blocks from the source.
    pdatas_all  = concatdPR $ map pnestedPData pdatas

    -- Cull the blocks from the source array so the
    -- SSegd only references the ones needed in the
    -- result.
    (ssegd_culled, pdatas_culled)
                = cullOnSSegd ssegd_all pdatas_all

    -- Build the final VSegd
    vsegd'      = promoteSSegd ssegd_culled
```

9. Barnes-Hut Kernel

This is the kernel of the Barnes-Hut benchmark described in §7 of the main paper.

```
-- A point with some mass.
data MassPoint   = MP  Double Double Double
--                       X      Y     mass

-- Acceleration vector.
type Accel       = (Double, Double)

-- Bounding box for points.
data BoundingBox = Box Double Double Double Double

-- The Barnes-Hut Quad-Tree
data BHTree
    = BHT Double          -- Size of box.
          Double          -- Centroid X.
          Double          -- Centroid Y.
          Double          -- Centroid mass.
          [:BHTree:]      -- Children.

-- | Given a bounding box containing all the points,
--   calculate their accelerations.
calcAccelsWithBox
    :: Double             -- Simulation smoothing param.
    -> BoundingBox -> [:MassPoint:] -> [:Accel:]

calcAccelsWithBox epsilon box points
 = [: calcAccel epsilon m tree | m <- points :]
 where tree = buildTree box points

-- | Build the Barnes-Hut quadtree tree.
buildTree :: BoundingBox -> [:MassPoint:] -> BHTree
buildTree bb points
 | lengthP points <= 1      = BHT s x y m emptyP
 | otherwise                = BHT s x y m subTrees
 where  MP x y m            = calcCentroid points
        (boxes, splitPnts)  = splitPoints bb points
        subTrees
             = [: buildTree bb' ps
                  | (bb', ps) <- zipP boxes splitPnts:]

        Box llx lly rux ruy = bb
        sx   = rux - llx
        sy   = ruy - lly
        s    = if sx < sy then sx else sy

-- | Split points according to their locations in
--   the quadrants.
splitPoints
        :: BoundingBox
        -> [: MassPoint :]
        -> ([:BoundingBox:], [:[: MassPoint :]:])

splitPoints b@(Box llx lly rux  ruy) points
  | noOfPoints <= 1 = (singletonP b, singletonP points)
  | otherwise
  = unzipP [: (b,p) | (b,p) <- zipP boxes splitPars
                    , lengthP p > 0:]
  where noOfPoints  = lengthP points
        lls         = [: p | p <- points, inBox b1 p :]
        lus         = [: p | p <- points, inBox b2 p :]
        rus         = [: p | p <- points, inBox b3 p :]
        rls         = [: p | p <- points, inBox b4 p :]
        b1          = Box llx  lly  midx midy
        b2          = Box llx  midy midx  ruy
        b3          = Box midx midy rux   ruy
        b4          = Box midx lly  rux  midy
        boxes       = [:b1,  b2,  b3,  b4:]
        splitPars   = [:lls, lus, rus, rls:]
        (midx,  midy) = ((llx + rux) / 2.0, (lly + ruy) / 2.0)

-- | Check if point is in box.
--   (excluding left and lower border)
inBox :: BoundingBox -> MassPoint -> Bool
inBox (Box llx  lly rux  ruy) (MP px  py  _)
 = (px > llx) && (px <= rux) && (py > lly) && (py <= ruy)


-- | Calculate the centroid of some points.
calcCentroid:: [:MassPoint:] -> MassPoint
calcCentroid mpts
 = MP  (sumP xs / mass) (sumP ys / mass) mass
 where
   mass     = sumP   [:m               | MP _ _ m  <- mpts:]
   (xs, ys) = unzipP [:(m * x, m * y) | MP x y m  <- mpts:]


-- | Calculate the acceleration of a point due to the
--    points in the given tree.
calcAccel :: Double
          -> MassPoint -> BHTree -> (Double, Double)

calcAccel epsilon point (BHT s x y m subtrees)
        | lengthP subtrees == 0
        = accel epsilon point (MP x y m)

        | isFar mpt s x y
        = accel epsilon point (MP x y m)

        | otherwise
        = let (xs, ys)
               = unzipP [: calcAccel epsilon point st
                         |  st <- subtrees :]
          in  (sumP xs, sumP ys)


-- | Calculate the acceleration between points.
accel   :: Double    -- Smoothing parameter.
        -> MassPoint -- The point being accelerated.
        -> MassPoint -- Neighbouring point.
        -> Accel

accel epsilon (MP x1 y1 _) (MP x2 y2 m)
 = (aabs * dx / r , aabs * dy / r)
 where  rsqr = (dx * dx) + (dy * dy) + epsilon * epsilon
        r    = sqrt rsqr
        dx   = x1 - x2
        dy   = y1 - y2
        aabs = m  / rsqr


-- | If the point is far from a box in the tree then we
--   can use its centroid as an approximation of all the
--   points in the corresponding branch.
isFar   :: MassPoint  -- Point being accelerated.
        -> Double     -- Size of box.
        -> Double     -- X pos of centroid.
        -> Double     -- Y pos of centroid.
        -> Bool

isFar (MP x1 y1 m) s x2 y2
 = let  dx      = x2 - x1
        dy      = y2 - y1
        dist    = sqrt (dx * dx + dy * dy)
   in   (s / dist) < 1
```