Optimizing `smallpt`

Davean Scies, Siddharth Bhat

Haskell Exchange

November 4th, 2020

Optimizing `smallpt`

2020-11-05

Optimizing `smallpt`

Davean Scies, Siddharth Bhat

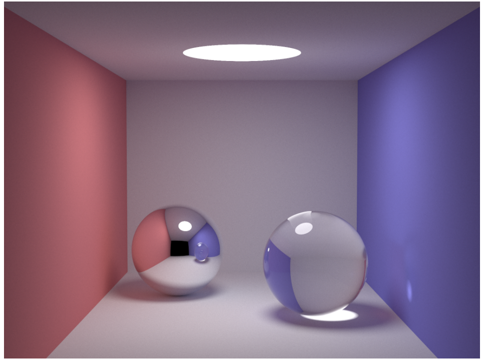
Haskell Exchange

November 4th, 2020

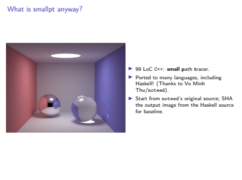
What is smallpt anyway?

1. **S**
2. 99 LoC `C++`: **small p**ath **t**racer.
3. Ported to many languages, including Haskell!
4. Haskell port was by Vo Minh Thu. Thanks a ton!
5. Start from `noteed`'s original source; SHA the output image from the Haskell source for baseline.
6. Perfect for an optimization case study.
7. Plan: Quick walk through Haskell code, end up at `C++` (`clang++`) performance.

What is smallpt anyway?



- 99 LoC `C++`: **small** **p**ath **t**racer.
- Ported to many languages, including Haskell! (Thanks to Vo Minh Thu/`noteed`).
- Start from `noteed`'s original source; SHA the output image from the Haskell source for baseline.



1. S
2. 99 LoC `C++`: **small** **p**ath **t**racer.
3. Ported to many languages, including Haskell!
4. Haskell port was by Vo Minh Thu. Thanks a ton!
5. Start from `noteed`'s original source; SHA the output image from the Haskell source for baseline.
6. Perfect for an optimization case study.
7. Plan: Quick walk through Haskell code, end up at `C++` (`clang++`) performance.

What is smallpt anyway?

```
struct Vec {
  double x, y, z; // position, also color (r,g,b)
  ... methods...
};
struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };
enum Refl_t { DIFF, SPEC, REFR };  // material types, used in radiance()
struct Sphere {
  double rad;    // radius
  Vec p, e, c;  // position, emission, color
  Refl_t refl;  // reflection type (DIFFuse, SPECular, REFRactive)
  ... methods ...
  double intersect(const Ray &r) const // returns distance, 0 if nohit
};
Sphere spheres[] = {//Scene: radius, position, emission, color, material
  Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),//Left
  ... initialization ...
};
```


What is smallpt anyway?

```
struct Vec {
  double x, y, z; // position, also color (r,g,b)
  ... methods...
};
struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };
enum Refl_t { DIFF, SPEC, REFR };  // material types, used in radiance()
struct Sphere {
  double rad;    // radius
  Vec p, e, c;  // position, emission, color
  Refl_t refl;  // reflection type (DIFFuse, SPECular, REFRactive)
  ... methods ...
  double intersect(const Ray &r) const // returns distance, 0 if nohit
};
Sphere spheres[] = {//Scene: radius, position, emission, color, material
  Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),//Left
  ... initialization ...
};
```


1. S
2. Has geometric primitives: vectors, spheres, materials
3. Entirely number-based, no real data structure

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){









}
```


What is smallpt anyway?

Vec radiance(const Ray &r, int depth, unsigned short *Xi){
}

1. **S**
2. Most of the compute cost is spent in the function that traces rays.
3. is called radiance

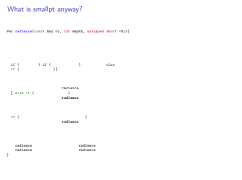
What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){









                        radiance

                        radiance




                        radiance




    radiance                    radiance
    radiance                    radiance
}
```


1. **S**
2. Recursively calls itself a bunch of times

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){





  if (          ) if (               )            else
  if (                 ){




  } else if (                radiance
                                )
                             radiance




  if (                                    )
                             radiance




    radiance                          radiance
    radiance                          radiance
}
```

1. **S**
2. Recursion is guarded by a lot of control flow

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){



  Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;

  if (          ) if (               )             else
  if (                 ){




                          radiance
  } else if (                 )
                          radiance




  if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)
                          radiance





    radiance                          radiance
    radiance                          radiance
}
```


1. S
2. The control flow and computation is very numeric in nature

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){



  Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;

  if (         ) if (erand48(Xi)  )            else
  if (                )){
                    erand48(Xi)     erand48(Xi)



                         radiance
  } else if (               )
                         radiance



  if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)
                         radiance




                                 erand48(Xi)
    radiance                    radiance
    radiance                    radiance
}
```


1. S
2. We use erand48 for randomness

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
  double t;                               // distance to intersection
  int id=0;                               // id of intersected object
  if (!intersect(r, t, id)) return Vec(); // if miss, return black
  const Sphere &obj = spheres[id];        // the hit object
  Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
  double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
  if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
  if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
    double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
    Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
    Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
    return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
  } else if (obj.refl == SPEC)            // Ideal SPECULAR reflection
    return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
  Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFRACTION
  bool into = n.dot(nl)>0;                // Ray from outside going in?
  double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
  if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
    return obj.e + f.mult(radiance(reflRay,depth,Xi));
  Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
  double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
  double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
  return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
    radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
    radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
}
```


1. S
2. The full code continues to be more of the same

Initial Haskell Code: `radiance` (1×)

```
radiance :: Ray -> CInt -> Ptr CUShort -> IO Vec
radiance ray@(Ray o d) depth xi = case intersects ray of
  (Nothing,_) -> return zerov
  (Just t,Sphere _r p e c refl) -> do




        continue f = case refl of - BRANCHING
          DIFF -> do
            r1 <- ((2*pi)*) `fmap` erand48 xi -- RNG







                 radiance

          SPEC -> do

            rad <- radiance   -- RECURSION

          REFR -> do







            if
              then do
                rad <- radiance
            ...
```


1. S
2. the original source has the same computation in haskell

Initial Haskell Code: Data structures

```haskell
data Vec = Vec {-# UNPACK #-} !Double {-# UNPACK #-} !Double {-# UNPACK #-} !Double

cross :: Vec -> Vec -> Vec
(.*) :: Vec -> Double -> Vec
infixl 7 .*
len :: Vec -> Double
norm :: Vec -> Vec
norm v = v .* recip (len v)
dot :: Vec -> Vec -> Double
maxv :: Vec -> Double

data Ray = Ray Vec Vec -- origin, direction
data Refl = DIFF | SPEC | REFR -- material types, used in radiance
-- | radius, position, emission, color, reflection
data Sphere = Sphere Double Vec Vec Vec Refl
```


1. **S**
2. We implement the same data structures in Haskell
3. Note that `Ray`, `Sphere` not having unpack

Initial Haskell Code: scene data

```
spheres :: [Sphere]
spheres =
  [ Sphere 1e5  (Vec (1e5+1) 40.8 81.6)  0 (Vec 0.75 0.25 0.25) DIFF --Left
  , Sphere 1e5  (Vec (99-1e5) 40.8 81.6) 0 (Vec 0.25 0.25 0.75) DIFF --Rght
  , Sphere 1e5  (Vec 50 40.8 1e5)        0 0.75  DIFF --Back
  , Sphere 1e5  (Vec 50 40.8 (170-1e5))  0 0     DIFF --Frnt
  , Sphere 1e5  (Vec 50 1e5 81.6)        0 0.75  DIFF --Botm
  , Sphere 1e5  (Vec 50 (81.6-1e5) 81.6) 0 0.75  DIFF --Top
  , Sphere 16.5 (Vec 27 16.5 47)         0 0.999 SPEC --Mirr
  , Sphere 16.5 (Vec 73 16.5 78)         0 0.999 REFR --Glas
  , Sphere 600  (Vec 50 681.33 81.6)    12 0     DIFF]--Lite
```


1. S
2. this list will be walked many times, as it contains our scene information.

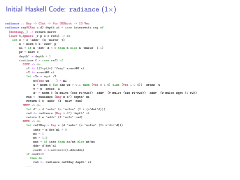
Initial Haskell code: Sphere intersection

```haskell
intersect :: Ray -> Sphere -> Maybe Double
intersect (Ray o d) (Sphere r p _e _c _refl) =
  if det<0 then Nothing else f (b-sdet) (b+sdet)
  where op = p - o -- Numeric
        eps = 1e-4
        b = dot op d
        det = b*b - dot op op + r*r -- Numeric
        sdet = sqrt det
        f a s = if a>eps then Just a else if s>eps then Just s else Nothing
intersects :: Ray -> (Maybe Double, Sphere)
intersects ray = (k, s)
  where (k,s) = foldl' f (Nothing,undefined) spheres -- Spheres iterated over
        f (k',sp) s' = case (k',intersect ray s') of
                    (Nothing,Just x) -> (Just x,s')
                    (Just y,Just x) | x < y -> (Just x,s')
                    _ -> (k',sp)
```


1. S
2. Responsible for figuring out what the ray hits.
3. We iterate over the list of spheres.
4. Once again, numeric heavy.
5. Use a Maybe to indicate whether we've found an answer or not.

Initial Haskell Code: radiance (1×)

```haskell
radiance :: Ray -> CInt -> Ptr CUShort -> IO Vec
radiance ray@(Ray o d) depth xi = case intersects ray of
  (Nothing,_) -> return zerov
  (Just t,Sphere _r p e c refl) -> do
    let x = o `addv` (d `mulvs` t)
        n = norm $ x `subv` p
        nl = if n `dot` d < 0 then n else n `mulvs` (-1)
        pr = maxv c
        depth' = depth + 1
        continue f = case refl of
          DIFF -> do
            r1 <- ((2*pi)*) `fmap` erand48 xi
            r2 <- erand48 xi
            let r2s = sqrt r2
                w@(Vec wx _ _) = nl
                u = norm $ (if abs wx > 0.1 then (Vec 0 1 0) else (Vec 1 0 0)) `cross` w
                v = w `cross` u
                d' = norm $ (u`mulvs`(cos r1*r2s)) `addv` (v`mulvs`(sin r1*r2s)) `addv` (w`mulvs`sqrt (1-r2))
            rad <- radiance (Ray x d') depth' xi
            return $ e `addv` (f `mulv` rad)
          SPEC -> do
            let d' = d `subv` (n `mulvs` (2 * (n`dot`d)))
            rad <- radiance (Ray x d') depth' xi
            return $ e `addv` (f `mulv` rad)
          REFR -> do
            let reflRay = Ray x (d `subv` (n `mulvs` (2* n`dot`d)))
                into = n`dot`nl > 0
                nc = 1
                nt = 1.5
                nnt = if into then nc/nt else nt/nc
                ddn= d`dot`nl
                cos2t = 1-nnt*nnt*(1-ddn*ddn)
            if cos2t<0
              then do
                rad <- radiance reflRay depth' xi
            ...
```

1. S
2. Branch heavy
3. Recursive
4. Uses an RNG

Initial Haskell Code: Entry point (1×)

```haskell
smallpt :: Int -> Int -> Int -> IO ()
smallpt w h nsamps = do
  ...
  c <- VM.replicate (w * h) 0
  allocaArray 3 \xi -> -- Create mutable memory
    flip mapM_ [0..h-1] $ \y -> do -- Loop
      writeXi xi y
      for_ [0..w-1] \x -> do -- Loop
        let i = (h-y-1) * w + x
        for_ [0..1] \sy -> do -- Loop
          for_ [0..1] \sx -> do -- Loop
            r <- newIORef 0 -- Create mutable memory
            for_ [0..samps-1] \_s -> do -- Loops, Loops
              r1 <- (2*) <$> erand48 xi
              ...
              rad <- radiance (Ray (org+d.*140) (norm d)) 0 xi  -- Crunch
              ...
              modifyIORef r (+ rad .* recip (fromIntegral samps)) -- Write
            ci <- VM.unsafeRead c i
            Vec rr rg rb <- readIORef r
            VM.unsafeWrite c i $
                ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25 -- Write
            ...
```


1. S
2. Uses mutability
3. Performs number crunchy loops
4. Finally, writes results out

Initial Haskell Code: File I/O (1×)

```
withFile "image.ppm" WriteMode $ \hdl -> do
      hPrintf hdl "P3\n%d %d\n%d\n" w h (255::Int)
      flip mapM_ [0..w*h-1] \i -> do
        Vec r g b <- VM.unsafeRead c i
        hPrintf hdl "%d %d %d " (toInt r) (toInt g) (toInt b)
```


Initial Haskell Code: File I/O (1×)

```
withFile "image.ppm" WriteMode $ \hdl -> do
      hPrintf hdl "P3\n%d %d\n%d\n" w h (255::Int)
      flip mapM_ [0..w*h-1] \i -> do
        Vec r g b <- VM.unsafeRead c i
        hPrintf hdl "%d %d %d " (toInt r) (toInt g) (toInt b)
```

Initial Haskell Code: RNG (1×)

```haskell
foreign import ccall unsafe "erand48"
  erand48 :: Ptr CUShort -> IO Double
```


Initial Haskell Code: RNG (1×)

foreign import ccall unsafe "erand48"
erand48 :: Ptr CUShort -> IO Double

1. **S**
2. We use the RNG to decide randomly in which direction to send rays
3. Point out the use of foreign CCall.

Restrict export list to ‘main‘ (1.13×)

```
-module Main where
+module Main (main) where
```


Restrict export list to ‘main‘ (1.13×)

-module Main where
+module Main (main) where

1. S
2. The very first thing to do is to let the compiler actually optimize.
3. Compiler can’t know how exported functions are used.
4. Export lists not just about encapsulation.

Restrict export list to 'main' (1.13×)

```
-module Main where
+module Main (main) where
```

- Exported functions could be used by something unknown.
- Original versions must be available.

Restrict export list to 'main' (1.13×)

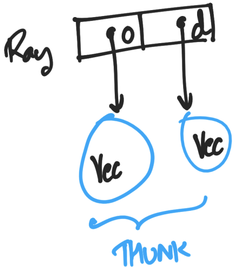
-module Main where
+module Main (main) where

- Exported functions could be used by something unknown.
- Original versions must be available.

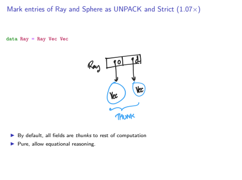
1. S
2. The very first thing to do is to let the compiler actually optimize.
3. Compiler can't know how exported functions are used.
4. Export lists not just about encapsulation.

Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

```
data Ray = Ray Vec Vec
```

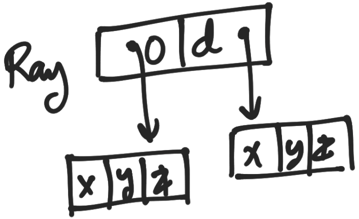


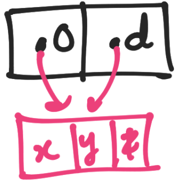
- By default, all fields are *thunks* to rest of computation
- Pure, allow equational reasoning.



Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

```
data Ray = Ray !Vec !Vec
```





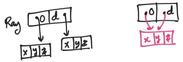
- When strict, elements are *pointers* to known structures
- pointers enable sharing!

2020-11-05

Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

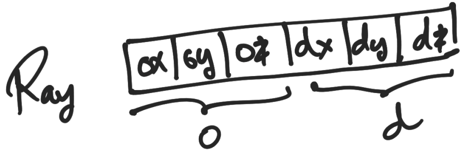
data Ray = Ray !Vec !Vec



- When strict, elements are *pointers* to known structures
- pointers enable sharing!

Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

```
data Ray = Ray {-# UNPACK #-} !Vec {-# UNPACK #-} !Vec
```



- When unpacked, elements are *members* of the parent.
- Larger, but eliminate pointer chasing.

Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

data Ray = Ray {-# UNPACK #-} !Vec {-# UNPACK #-} !Vec

- When unpacked, elements are *members* of the parent.
- Larger, but eliminate pointer chasing.

Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

```
data Vec = Vec {-# UNPACK #-} !Double
               {-# UNPACK #-} !Double
               {-# UNPACK #-} !Double

-data Ray = Ray Vec Vec -- origin, direction
+data Ray = Ray !Vec !Vec -- origin, direction

 data Refl = DIFF | SPEC | REFR -- material types, used in radiance

 -- radius, position, emission, color, reflection
-data Sphere = Sphere Double Vec Vec Vec !Refl
+data Sphere = Sphere {-# UNPACK #-} !Double
+                     {-# UNPACK #-} !Vec
+                     {-# UNPACK #-} !Vec
+                     {-# UNPACK #-} !Vec !Refl

struct Vec { double x, y, z; }
struct Ray { std::function<Vec()> v; std::function<Vec()> w; };
struct RayUnpack { double xv, yv, int zv;
                   double xw, yw, zw; };
```


1. **D**
2. Strictness in the arguments means that they're evaluated when instantiated, not when demanded.
3. Where as Unpacking removes indirection from doing a memory lookup for components.
4. Means we have to copy everything into the data structure that it is unpacked into.
5. We don't unpack ray (Lots of calculations on its components, want those to fuse)
6. Unpack Sphere - its static from compile time
7. Don't unpack Ray, because each Vec undergoes a lot of computation.

Use a pattern synonym to unpack Refl in Sphere (1.07×)

```
+{-# LANGUAGE PatternSynonyms #-}

-data Refl = DIFF | SPEC | REFR -- material types, used in radiance
+newtype Refl = Refl Int  -- material types, used in radiance
+pattern DIFF,SPEC,REFR :: Refl
+pattern DIFF = Refl 0
+pattern SPEC = Refl 1
+pattern REFR = Refl 2
+{-# COMPLETE DIFF, SPEC, REFR #-}

 -- radius, position, emission, color, reflection
 data Sphere = Sphere {-# UNPACK #-} !Double
                      {-# UNPACK #-} !Vec
                      {-# UNPACK #-} !Vec
-                     {-# UNPACK #-} !Vec !Refl
+                     {-# UNPACK #-} !Vec {-# UNPACK #-} !Refl
```


1. **D**
2. Was unable to unpack Refl
3. UnboxedSums are recent
4. UnboxedSums are very unpleasant
5. We're using an older trick to fake the unboxing here instead.
6. In this case it isn't much of a win, but it illustrates the technique.

Change from `maximum` on a list to `max` (1.08×)

```
-maxv (Vec a b c) = maximum [a,b,c]
+maxv (Vec a b c) = max a (max b c)

     let x = o `addv` (d `mulvs` t)
         n = norm $ x `subv` p
         nl = if n `dot` d < 0 then n else n `mulvs` (-1)
-        pr = maxv c
         depth' = depth + 1
         continue f = case refl of
           DIFF -> do
...
     if depth'>5
       then do
         er <- erand48 xi
+        let !pr = maxv c
```


Change from maximum on a list to max (1.08×)

```
-maxv (Vec a b c) = maximum [a,b,c]
+maxv (Vec a b c) = max a (max b c)

     let x = o `addv` (d `mulvs` t)
         n = norm $ x `subv` p
         nl = if n `dot` d < 0 then n else n `mulvs` (-1)
-        pr = maxv c
         depth' = depth + 1
         continue f = case refl of
           DIFF -> do
...
     if depth'>5
       then do
         er <- erand48 xi
+        let !pr = maxv c
```


1. **D**
2. Prebuild comparison
3. Don't go via list
4. GHC does not evaluate at compile time, only has RULES
5. Doesn't really help much in this case

Convert erand48 to pure Haskell (1.09×)

```
-foreign import ccall unsafe "erand48"
-  erand48 :: Ptr CUShort -> IO Double

+erand48 :: IORef Word64 -> IO Double
+erand48 !t =  do -- | Some number crunchy thing.
+  r <- readIORef t
+  let x' = 0x5deece66d * r + 0xb
+      d_word = 0x3ff0000000000000 .|. ((x' .&. 0xffffffffffff) `unsafeShiftL` 4)
+      d = castWord64ToDouble d_word - 1.0
+  writeIORef t x'
+  pure d
...
-radiance :: Ray -> CInt -> Ptr CUShort -> IO Vec
+radiance :: Ray -> Int -> IORef Word64 -> IO Vec -- IORef with state
 radiance ray@(Ray o d) depth xi = case intersects ray of
   ...
   c <- VM.replicate (w * h) zerov
-  allocaArray 3 $ \xi -> -- Old RNG state
-      flip mapM_ [0..h-1] $ \y -> do
+  xi <- newIORef 0 -- New RNG state
+  flip mapM_ [0..h-1] $ \y -> do
      writeXi xi y
```


1. S
2. The entire premise of this talk is that Haskell can be as fast as C.
3. We're opening the black box of what erand48 does to GHC
4. Further any impedance mismatch, such as FFI almost universally has to have, carries some bookkeeping overhead.
5. If our Haskell code was as fast as the C code moving the code into Haskell would be a win, if it was slightly slower it could still be a win.
6. Often considering your Haskell code's performance is a better option and easier than reimplementing something in C.
7. As is the way with optimizations, this is not universally true.

Remove mutability: Erand48 Monad

```
-erand48 :: IORef Word64 -> IO Double
-erand48 !t =  do
-  r <- readIORef t
+data ET a = ET !Word64 !a deriving Functor
+newtype Erand48 a = Erand48 { runErand48' :: Word64 -> ET a } deriving Functor
+instance Applicative Erand48 where
+instance Monad Erand48 where
+runWithErand48 :: Int -> Erand48 a -> a
+erand48 :: Erand48 Double
...
-radiance :: Ray -> Int -> IORef Word64 -> IO Vec
-radiance ray@(Ray o d) depth xi = case intersects ray of
+radiance :: Ray -> Int -> Erand48 Vec
+radiance ray@(Ray o d) depth = case intersects ray of
...
-            r1 <- (2*pi*) <$> erand48 xi
-            r2 <- erand48 xi
+            r1 <- (2*pi*) <$> erand48
+            r2 <- erand48
...
-                              then (.* rp) <$> radiance reflRay depth' xi
-                              else (.* tp) <$> radiance (Ray x tdir) depth' xi
+                              then (.* rp) <$> radiance reflRay depth'
+                              else (.* tp) <$> radiance (Ray x tdir) depth'
```


1. **D**
2. All these mutability locations throw in extra RTS code, extra sequencing that blocks the compiler's optimization, and dependency chains.
3. Sometimes we need mutability for performance.
4. SSA is normal to compilers though.
5. We almost start at SSA as a functional language.
6. don't break it when you don't have a good reason.

Removing mutation: eliminate IORef and Data.Vector.Mutable

```
-  c <- VM.replicate (w * h) 0
-  xi <- newIORef 0
-  flip mapM_ [0..h-1] $ \y -> do
-      writeXi xi y
-      for_ [0..w-1] \x -> do
-        let i = (h-y-1) * w + x
-        for_ [0..1] \sy -> do
-          for_ [0..1] \sx -> do
-            r <- newIORef 0
-            for_ [0..samps-1] \_s -> do
-              r1 <- (2*) <$> erand48 xi
+      img = (`concatMap` [(h-1),(h-2)..0]) $ \y -> runWithErand48 y do
+        for [0..w-1] \x -> do
+          (\pf -> foldlM pf 0 [(sy, sx) | sy <- [0,1], sx <- [0,1]]) \ci (sy, sx) -> do
+            Vec rr rg rb <- (\f -> foldlM f 0 [0..samps-1]) \ !r _s -> do
+              r1 <- (2*) <$> erand48
...
-              modifyIORef r (+ rad .* recip (fromIntegral samps))
-            ci <- VM.unsafeRead c i
-            Vec rr rg rb <- readIORef r
-            VM.unsafeWrite c i $ ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25
+              pure (r + rad .* recip (fromIntegral samps))
+            pure (ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25)
...
```


1. D

Set **everything** in `smallpt` to be strict (1.17×)

```
smallpt :: Int -> Int -> Int -> IO ()
smallpt w h nsamps = do
-  let samps = nsamps `div` 4
-      org = Vec 50 52 295.6
-      dir = norm $ Vec 0 (-0.042612) (-1)
-      cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
-      cy = norm (cx `cross` dir) `mulvs` 0.5135
+  let !samps = nsamps `div` 4
+      !org = Vec 50 52 295.6
+      !dir = norm $ Vec 0 (-0.042612) (-1)
+      !cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
+      !cy = norm (cx `cross` dir) `mulvs` 0.5135
...
-  r1 <- (2*) `fmap` erand48 xi
-  let dx = if r1<1 then sqrt r1-1 else 1-sqrt(2-r1)
-  r2 <- (2*) `fmap` erand48 xi
-  let dy = if r2<1 then sqrt r2-1 else 1-sqrt(2-r2)
-      d = ...
-  rad <- radiance (Ray (org`addv`(d`mulvs`140)) (norm d)) 0 xi
+  !r1 <- (2*) `fmap` erand48 xi
+  let !dx = if r1<1 then sqrt r1-1 else 1-sqrt(2-r1)
+  !r2 <- (2*) `fmap` erand48 xi
+  let !dy = if r2<1 then sqrt r2-1 else 1-sqrt(2-r2)
+      !d = ...
...
+              pure $! r + rad .* recip (fromIntegral samps)
+            pure $! ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25
```


Don't senselessly bang everything in sight.

1. **D**
2. This is not a recommendation, this is a warning.
3. We get a speedup here but it can also regress performance. Some of these bangs are regressions that are hidden.

Why strictness may be bad

```
let foo = let x = error "ERR" in \y -> y
let fooOpt = \y -> y
```


Why strictness may be bad

let foo = let x = error "ERR" in \y -> y
let fooOpt = \y -> y

1. **S**
2. Consider the function foo and fooOpt. These are equivalent
3. The fact that x is not used allows us to eliminiate computing x
4. Consider the next version
5. Illegal, we need to have x, because it doesn't produce ERR
6. we can't equationally reason about the program anymore.
7. Makes it harder for GHC. GHC is conservative about bangs
8. Inhibits compiler from optimizing

Why strictness may be bad

```
let foo = let x = error "ERR" in \y -> y
let fooOpt = \y -> y

let foo' = let !x = error "ERR" in \y -> y
```


Why strictness may be bad

let foo = let x = error "ERR" in \y -> y
let fooOpt = \y -> y
let foo' = let !x = error "ERR" in \y -> y

1. **S**
2. Consider the function foo and fooOpt. These are equivalent
3. The fact that x is not used allows us to eliminiate computing x
4. Consider the next version
5. Illegal, we need to have x, because it doesn't produce ERR
6. we can't equationally reason about the program anymore.
7. Makes it harder for GHC. GHC is conservative about bangs
8. Inhibits compiler from optimizing

Why strictness may be bad

```haskell
let foo = let x = error "ERR" in \y -> y
let fooOpt = \y -> y

let foo' = let !x = error "ERR" in \y -> y

let foo'Opt = \y -> y  -- ERROR! forcing foo'=foo'Opt should give "ERR"
```


Why strictness may be bad

let foo = let x = error "ERR" in \y -> y
let fooOpt = \y -> y
let foo' = let !x = error "ERR" in \y -> y
let foo'Opt = \y -> y -- ERROR! forcing foo'=foo'Opt should give "ERR"

1. S
2. Consider the function foo and fooOpt. These are equivalent
3. The fact that x is not used allows us to eliminiate computing x
4. Consider the next version
5. Illegal, we need to have x, because it doesn't produce ERR
6. we can't equationally reason about the program anymore.
7. Makes it harder for GHC. GHC is conservative about bangs
8. Inhibits compiler from optimizing

Reduce to only useful strictnesses in smallpt(1.17×)

```
-  let !samps = nsamps `div` 4
-      !org = Vec 50 52 295.6
-      !dir = norm $ Vec 0 (-0.042612) (-1)
-      !cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
-      !cy = norm (cx `cross` dir) .* 0.5135
+  let samps = nsamps `div` 4
+      org = Vec 50 52 295.6
+      dir = norm $ Vec 0 (-0.042612) (-1)
+      cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
+      cy = norm (cx `cross` dir) .* 0.5135
...
-              !r1 <- (2*) <$> erand48
+              r1 <- (2*) <$> erand48
...
-              !r2 <- (2*) <$> erand48
+              r2 <- (2*) <$> erand48
...
-              !rad <- radiance (Ray (org+d.*140) (norm d)) 0
+              rad <- radiance (Ray (org+d.*140) (norm d)) 0
...
-              pure $! r + rad .* recip (fromIntegral samps)
-            pure $! ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25
+              pure (r + rad .* recip (fromIntegral samps))
+            pure (ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25)
```


Reduce to only useful strictnesses in smallpt(1.17×)

1. D
2. Thus the compiler can no longer move the computation around or simplify it.
3. Force useless work.
4. A little thinking about how the variables are used or looking at core allows us to select which ones we bang selectively.

Use strictness strategically in entire project

```
...
-  if det<0 then Nothing else f (b-sdet) (b+sdet)
-  where op = p - o
-        eps = 1e-4
-        b = dot op d
-        det = b*b - dot op op + r*r
-        sdet = sqrt det
-        f a s = if a>eps then Just a else if s>eps then Just s else Nothing
+  if det<0
+  then Nothing
+  else
+    let !eps = 1e-4
+        !sdet = sqrt det
+        !a = b-sdet
+        !s = b+sdet
+    in if a>eps then Just a else if s>eps then Just s else Nothing
...
```


Use strictness strategically in entire project

1. **D**
2. Sometimes (point out 'intersect') we have to rearrange the code though when we use bangs.
3. Bangs tell the compiler to make more efficient code, but take away the compiler's options in how to do so.
4. Only take away the compiler's liberties when it's using them poorly.
5. Becomes intuitive.

Remove Maybe from intersect(s) (1.32×)

```
| Old: Use Maybe Double to represent (was-hit?:bool, hit-distance: Double)
| New: use (1/0) to represent not (was-hit?)
-intersect :: Ray -> Sphere -> Maybe Double
+intersect :: Ray -> Sphere -> Double
intersect (Ray o d) (Sphere r p _e _c _refl) =
-  if det<0 then Nothing else f (b-sdet) (b+sdet)
+  if det<0 then (1/0.0) else f (b-sdet) (b+sdet)
   where op = p `subv` o
         ...
-        f a s = if a>eps then Just a else if s>eps then Just s else Nothing
+        f a s = if a>eps then a else if s>eps then s else (1/0.0)

-intersects :: Ray -> (Maybe Double, Sphere)
+intersects :: Ray -> (Double, Sphere)
 intersects ray = (k, s)
-  where (k,s) = foldl' f (Nothing,undefined) spheres
-        f (k',sp) s' = case (k',intersect ray s') of
-                  (Nothing,Just x) -> (Just x,s')
-                  (Just y,Just x) | x < y -> (Just x,s')
-                  _ -> (k',sp)
+  where (k,s) = foldl' f (1/0.0,undefined) spheres
+        f (k', sp) s' = let !x = intersect ray s' in if x < k' then (x, s') else (k', sp)

 radiance :: Ray -> Int -> STRefU s Word64 -> ST s Vec
 radiance ray@(Ray o d) depth xi = case intersects ray of
-  (Nothing,_) -> return zerov
-  (Just t,Sphere _r p e c refl) -> do
+  (t,_) | t == (1/0.0) -> return zerov
+  (t,Sphere _r p e c refl) -> do
```


1. S
2. This is a far more performance critical version of what we saw with 'maximum' vs. 'max'.
3. innermost functions are of critical importance. remove Maybe which significantly reduces the boxing
4. Since a Ray that fails to intersect something can be said to intersect at infinity, Double already actually covers the structure at play
5. This also reduces allocation.

Hand unroll the fold in intersects (1.35×)

```
intersects :: Ray -> (Double, Sphere)
-intersects ray = (k, s)
-  where (k,s) = foldl' f (1/0.0,undefined) spheres
+intersects ray =
+     f (... (f (f (intersect ray sphLeft, sphLeft) sphRight) ...)
+  where
     f (k', sp) s' = let !x = intersect ray s' in if x < k' then (x, s') else (k', sp)

-spheres :: [Sphere]
-spheres = let s = Sphere ; z = zerov ; (.*) = mulvs ; v = Vec in
-   [ s 1e5 (v (1e5+1) 40.8 81.6)     z (v 0.75 0.25 0.25) DIFF --Left
-   , s 1e5 (v (-1e5+99) 40.8 81.6)  z (v 0.25 0.25 0.75) DIFF --Rght
...

+sphLeft, sphRight, ...  :: Sphere
+sphLeft  = Sphere 1e5  (Vec (1e5+1) 40.8 81.6)    zerov (Vec 0.75 0.25 0.25) DIFF
+sphRight = Sphere 1e5  (Vec (-1e5+99) 40.8 81.6)  zerov (Vec 0.25 0.25 0.75) DIFF
...
```


Hand unroll the fold in intersects (1.35×)

1. **D**
2. 'intersects' is very hot
3. Loop unrolling
4. Many compilers do this for us, and there are special versions of it like Duff's Device.
5. Sadly GHC doesn't
6. Can do variants by hand.
7. RULE could handle each one specificly (only exactly that one)?

Custom datatype for `intersects` parameter passing

```
Old: Tuple with possibly-uenevaluated Double and Sphere
New: Reference to a guaranteed-to-be-evaluated Double and Sphere
-intersects :: Ray -> (Double, Sphere)
+data T = T !Double !Sphere
+
+intersects :: Ray -> T
 intersects ray =
-    f ( ... f (intersect ray sphLeft, sphLeft) sphRight) ... sphLite
+    f ( ... f (T (intersect ray sphLeft) sphLeft) sphRight) ... sphLite
   where
-    f (k', sp) s' =
-        let !x = intersect ray s' in if x < k' then (x, s') else (k', sp)
+    f !(T k' sp) !s' =
+        let !x = intersect ray s' in if x < k' then T x s' else T k' sp

 radiance :: Ray -> Int -> Erand48 Vec
 radiance ray@(Ray o d) depth = case intersects ray of
-  (!t,_) | t == 1/0.0 -> return 0
-  (!t,!Sphere _r p e c refl) -> do
+  (T t _) | t == 1/0.0 -> return 0
+  (T t (Sphere _r p e c refl)) -> do
    let !x = o + d .* t
        !n = norm $ x - p
        !nl = if dot n d < 0 then n else negate n
```


1. **D**
2. We can optimize data passing.
3. Want: Data strict, but not unpacked.
4. Compiler knows its evaluated but no copying
5. A normal tuple lacks strictness information.
6. An unboxed tuple forces copying
7. Strict Tuple.
8. This exists in libraries of course, but we wanted to illustrate it.

Optimize file writing

```
   build-depends:
       base >= 4.12 && < 4.15
+    , bytestring ^>= 0.11

-toInt :: Double -> Int
-toInt x = floor $ clamp x ** recip 2.2 * 255 + 0.5
+toInt :: Double -> BB.Builder -- O(1) concatenation
+toInt x = BB.intDec (floor (clamp x ** recip 2.2 * 255 + 0.5)) <> BB.char8 ' '
...
   withFile "image.ppm" WriteMode $ \hdl -> do
-        hPrintf hdl "P3\n%d %d\n%d\n" w h (255::Int)
-        for_ img \(Vec r g b) -> do
-          hPrintf hdl "%d %d %d " (toInt r) (toInt g) (toInt b)
+        BB.hPutBuilder hdl $
+          BB.string8 "P3\n" <>  -- efficient builders for ASCII
+          BB.intDec w <> BB.char8 ' ' <> BB.intDec h <> BB.char8 '\n' <>
+          BB.intDec 255 <> BB.char8 '\n' <>
+          (mconcat $ fmap (\(Vec r g b) -> toInt r <> toInt g <> toInt b) img)
```


Optimize file writing

```
   build-depends:
       base >= 4.12 && < 4.15
+    , bytestring ^>= 0.11

-toInt :: Double -> Int
-toInt x = floor $ clamp x ** recip 2.2 * 255 + 0.5
+toInt :: Double -> BB.Builder -- O(1) concatenation
+toInt x = BB.intDec (floor (clamp x ** recip 2.2 * 255 + 0.5)) <> BB.char8 ' '
...
   withFile "image.ppm" WriteMode $ \hdl -> do
-        hPrintf hdl "P3\n%d %d\n%d\n" w h (255::Int)
-        for_ img \(Vec r g b) -> do
-          hPrintf hdl "%d %d %d " (toInt r) (toInt g) (toInt b)
+        BB.hPutBuilder hdl $
+          BB.string8 "P3\n" <>  -- efficient builders for ASCII
+          BB.intDec w <> BB.char8 ' ' <> BB.intDec h <> BB.char8 '\n' <>
+          BB.intDec 255 <> BB.char8 '\n' <>
+          (mconcat $ fmap (\(Vec r g b) -> toInt r <> toInt g <> toInt b) img)
```


1. **D**
2. Strings are inefficient
3. 'bytestring' has some efficient writing code, so we just convert to that for a modest gain.

Use LLVM backend (1.87×)

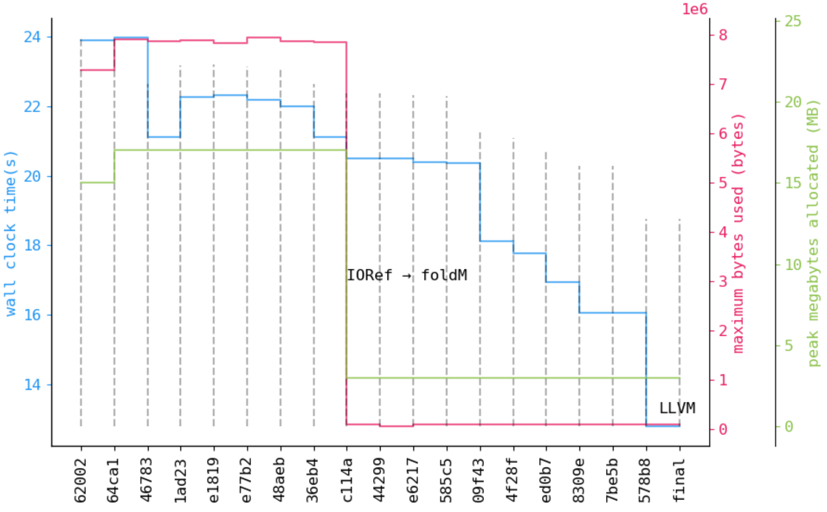
```
+package smallpt-opt
+   ghc-options: -fllvm
```


Use LLVM backend (1.87×)

+package smallpt-opt
+   ghc-options: -fllvm

1. S
2. Finally, this particular code is quite numeric heavy.
3. There are optimizations for numeric heavy code we're missing in GHC.
4. LLVM has an extensive library of laws to optimize low level numeric ops.
5. LLVM is too low-level to understand haskell as haskell.
6. LLVM makes decisions with the tacit assumption that the assembly came from a C-like language, which is often to the detriment of a Haskell-like language.
7. In this case, as the code is "fortran-like", LLVM wins.

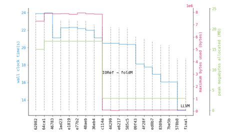
The view from the mountaintop



Optimizing `smallpt`

2020-11-05

The view from the mountaintop



1. D

Avoid CPU ieee754 slow paths

```
 intersect :: Ray -> Sphere -> Double
 intersect (Ray o d) (Sphere r p _e _c _refl) =
   if det<0
-  then 1/0.0
+  then 1e20
   else
    ...
-    in if a>eps then a else if s>eps then s else 1/0.0
+    in if a>eps then a else if s>eps then s else 1e20

...
 radiance :: Ray -> Int -> Erand48 Vec
 radiance ray@(Ray o d) depth = case intersects ray of
-  (T t _) | t == 1/0.0 -> return 0
+  (T 1e20 _) -> return 0
   ...
```


Avoid CPU ieee754 slow paths

```
 intersect :: Ray -> Sphere -> Double
 intersect (Ray o d) (Sphere r p _e _c _refl) =
   if det<0
-  then 1/0.0
+  then 1e20
   else
    ...
-    in if a>eps then a else if s>eps then s else 1/0.0
+    in if a>eps then a else if s>eps then s else 1e20

...
 radiance :: Ray -> Int -> Erand48 Vec
 radiance ray@(Ray o d) depth = case intersects ray of
-  (T t _) | t == 1/0.0 -> return 0
+  (T 1e20 _) -> return 0
   ...
```


1. **D**
2. We used +Inf to match the Maybeness
3. C++ code set 1e20 s the horizon
4. Mechanical sympathy is important.
5. Know how the CPU (abstractly) executes - slow path / fast path.

Fix differences with C++ version

```
-                    if depth>2
+                    if depth'>2 -- depth' = depth + 1
...
```


Fix differences with C++ version

```
-                    if depth>2
+                    if depth'>2 -- depth' = depth + 1
...
```


1. **S**
2. Since the sha1 of the output didn't match the C++ version we started investigating.
3. clang++, g++ actually produce different sha1s
4. unincremented depth was being used in one branch, causing us to do more work
5. now confident to say we're doing the same computation as C++

A second view from the mountaintop

2020-11-05

└─A second view from the mountaintop

A second view from the mountaintop

Takeaways

- The unrolling in 'intersects' is ugly.
- (We feel) the maintainability of this code hasn't been significantly harmed.
- We're faster than clang++ and within 6% of g++
- Haven't exhausted the optimization opportunities.
- GHC could learn to do several of these optimizations for us.
- Others are just good Haskell style.
- Clean Haskell is often performant Haskell.
- Repository stepping through each optimization is available at ...

Takeaways

- The unrolling in 'intersects' is ugly.
- (We feel) the maintainability of this code hasn't been significantly harmed.
- We're faster than clang++ and within 6% of g++
- Haven't exhausted the optimization opportunities.
- GHC could learn to do several of these optimizations for us.
- Others are just good Haskell style.
- Clean Haskell is often performant Haskell.
- Repository stepping through each optimization is available at ...

1. **D**

Raw data

- All test were on an otherwise idle `Equinix Metal c3.small.x86` (`Intel Xeon E-2278G` with `32GiB` RAM, Ubuntu `20.04`).

0e242	23.6586	23.658	23.7251	23.6954	23.676	23.656	23.673	23.7139	23.6598	23.7641
76a82	21.3573	21.384	21.3516	21.3304	21.3658	21.3776	21.3564	21.3843	21.3401	21.3683
f3819	22.1036	22.0754	22.1034	22.0864	22.0692	22.1004	22.0584	22.1181	22.0728	22.0972
fc364	22.1211	22.114	22.1205	22.1101	23.0621	22.1101	22.1163	22.133	22.1491	22.1464
66c39	21.8626	21.8684	21.8977	21.9043	21.893	21.8483	21.8335	21.8869	21.8848	21.8335
5704a	21.4682	21.4692	21.4411	21.473	21.4783	21.4613	21.4818	21.4507	21.4388	21.4933
eb9a3	20.6134	20.6014	20.6527	20.596	20.6034	20.6174	20.5965	20.594	20.5967	20.5892
5caf8	20.5209	20.535	20.5312	20.5289	20.5338	20.5717	20.5387	20.5386	20.5262	20.5488
361eb	20.5551	20.5485	20.5602	20.552	20.5668	20.555	20.5573	20.5564	20.5599	20.5823
7911c	19.2532	19.2664	19.2955	19.2565	19.2517	19.2722	19.3284	19.2611	19.2623	19.2596
149bc	16.8551	16.8551	16.8788	16.9067	16.8683	16.8685	16.8651	16.9186	16.8589	16.8604
80931	16.5752	16.5739	16.5831	16.5918	16.5678	16.5785	16.6128	16.5682	16.5816	16.577
07d68	16.1819	16.1672	16.1829	16.2267	16.1716	16.1854	16.1806	16.1949	16.1917	16.1784
4e141	16.2206	16.1816	16.2002	16.1799	16.1813	16.1781	16.1929	16.243	16.1705	16.1877
16c56	11.6334	11.6166	11.6837	11.6504	11.6227	11.6135	11.5949	11.5966	11.6013	11.64
d6347	11.1632	11.218	11.1741	11.1802	11.1849	11.1755	11.1729	11.2155	11.1718	11.2089
9ee06	10.2131	10.2154	10.1994	10.2105	10.2028	10.2344	10.2008	10.2497	10.2226	10.3042
gcc	10.97	10.97	10.97	10.99	10.98	10.97	10.97	10.98	10.98	10.98
clang	12.53	12.51	12.5	12.53	12.51	12.5	12.52	12.48	12.53	12.52