Optimizing smallpt

Davean Scies, Siddharth Bhat

Haskell Exchange

November 4th, 2020

Optimizing smallpt

2020-11-05

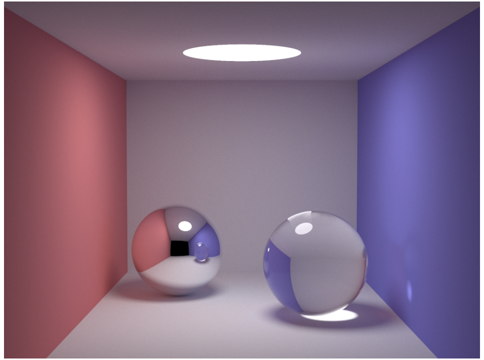
Optimizing smallpt

Davean Scies, Siddharth Bhat

Haskell Exchange

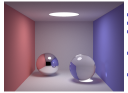
November 4th, 2020

What is smallpt anyway?



- 99 LoC `C++`: **small path tracer**.
- Perfect for an optimization case study.
- Ported to many languages, including Haskell! (Thanks to Vo Minh Thu/`noteed`).
- Start from `noteed`'s original source; SHA the output image for baseline, keep optimizing.
- Plan: Quick walk through Haskell code, end up at `C++` (`clang++`) performance.

What is smallpt anyway?



- 99 LoC `C++`: **small path tracer**.
- Perfect for an optimization case study.
- Ported to many languages, including Haskell! (Thanks to Vo Minh Thu/`noteed`).
- Start from `noteed`'s original source; SHA the output image for baseline, keep optimizing.
- Plan: Quick walk through Haskell code, end up at `C++` (`clang++`) performance.

Note for first slide: what is smallpt anyway?

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 ...
};
```


Its recursive

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){

}

Say that the core function is radiance

What is smallpt anyway?

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









                          radiance

                          radiance




                          radiance





    radiance                      radiance
    radiance                      radiance
}
```


Say that it's recursive

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
}
```


What is smallpt anyway?

...with 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 (                ){



  } else if (                radiance
                                )
                             radiance



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




    radiance                       radiance
    radiance                       radiance
}
```


and lots of arithmetic

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);
}
```


Talk about how dense the source is to fit in 99 lines.

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
          DIFF -> do








                   radiance

          SPEC -> do

            rad <- radiance

          REFR -> do







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


show the same thing, this time 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
```

Initial Haskell Code: scene data

```haskell
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
```

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)
```


Initial Haskell code: Sphere intersection

1. Draw attention the amount of numeric stuff
2. Draw attention the use of the list of spheres

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
            ...
```


Initial Haskell Code: radiance (1×)

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

Initial Haskell Code: File I/O (1×)

```haskell
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

Restrict export list to ‘main‘ (1.13×)

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


Restrict export list to ‘main‘ (1.13×)

-module Main where
+module Main (main) where

1. The very first thing to do is to let the compiler actually optimize.
2. Exported functions could be used by something unknown to the compiler and thus their original versions must be available. This makes many optimizations look bad or be unreasonable. Explicit export lists tell the compiler what you care about. They aren't just about encapsulation.

Restrict export list to ‘main‘ (1.13×)

```diff
-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. The very first thing to do is to let the compiler actually optimize.
2. Exported functions could be used by something unknown to the compiler and thus their original versions must be available. This makes many optimizations look bad or be unreasonable. Explicit export lists tell the compiler what you care about. They aren't just about encapsulation.

Restrict export list to ‘main‘ (1.13×)

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

- Exported functions could be used by something unknown.
- original versions must be available.
- Make many optimizations unsound.

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.
- Make many optimizations unsound.

1. The very first thing to do is to let the compiler actually optimize.
2. Exported functions could be used by something unknown to the compiler and thus their original versions must be available. This makes many optimizations look bad or be unreasonable. Explicit export lists tell the compiler what you care about. They aren't just about encapsulation.

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 {-# UNPACK #-} !Vec {-# UNPACK #-} !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; };
```

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

1. Strictness in the arguments means that they're evaluated when instantiated, not when demanded. This can save a bit in their evaluation cost but means we always pay it. It is a trade off but here we know it will be a good one because we have almost perfect demand.
2. Unpacking is different. It removes indirection from doing a memory lookup for components but means we have to copy everything into the data structure that it is unpacked into.
3. We don't unpack ray because we'll be doing a lot of calculations on it and we want the Vec math to fuse and not get copied, when we use the Vecs they should be hot in cache anyway.
4. We do inline Sphere because we statically create them and while they'll also be hot in cache there is no benefit to taking any extra cost at all.

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
```


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

1. Until the recent addition of unboxed sums in GHC there was no way to unpack a type with multiple constructors.
2. While unboxed sums exist now, their syntax is quite unpleasant.
3. We're using an older trick to fake the unboxing here instead.
4. In this case it isn't much of a win, but it illustrates the technique.

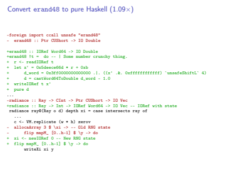
Change from `maximum` on a list to `max` (1.08×)

```
-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
```


1. finicky optimization. GHC does not evaluate at compile time, making optimizations like these necessary
2. This function isn't used much so it doesn't really impact our performance but often building unnecessary data structures is detrimental.

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. The entire premise of this talk is that Haskell can be as fast as C.
2. We're opening the black box of what erand48 does to GHC
3. Further any impedance mismatch, such as FFI almost universally has to have, carries some bookkeeping overhead.
4. 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.
5. Often considering your Haskell code's performance is a better option and easier than reimplementing something in C.
6. 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'
```


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'
```

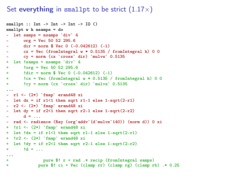

1. All these mutability locations throw in extra RTS code, extra sequencing that blocks the compiler's optimization, and dependency chains.
2. Sometimes we need mutability for performance.
3. HOWEVERm one primary optimization technique powering modern compilers is SSA.
4. We almost start there as a functional language.
5. 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)
...
```

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. This is not a recommendation, this is a warning.
2. 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

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

1. bangs remove freedom from the compiler
2. bangs force large objects to be evaluated
3. Haskell is awesome because equational reasoning.
4. Bangs are not great with equational reasoning.
5. judiciously use bangs. Too many remove choice from the compiler.

Why strictness may be bad

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

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


1. bangs remove freedom from the compiler
2. bangs force large objects to be evaluated
3. Haskell is awesome because equational reasoning.
4. Bangs are not great with equational reasoning.
5. judiciously use bangs. Too many remove choice from the compiler.

Why strictness may be bad

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

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

let tuple = (42, error "urk")
let bar = let (x, y) = tuple in \z -> x + z
```


Why strictness may be bad

let foo = let x = error "ERR" in \y -> y
let foo' = let !x = error "ERR" in \y -> y
let fooOpt = \y -> y
let foo'Opt = \y -> y -- ERROR! forcing foo' should give "ERR"
let tuple = (42, error "urk")
let bar = let (x, y) = tuple in \z -> x + z

1. bangs remove freedom from the compiler
2. bangs force large objects to be evaluated
3. Haskell is awesome because equational reasoning.
4. Bangs are not great with equational reasoning.
5. judiciously use bangs. Too many remove choice from the compiler.

Why strictness may be bad

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

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

let tuple = (42, error "urk")
let bar = let (x, y) = tuple in \z -> x + z

let barOpt = \z -> 42 + z
```


Why strictness may be bad

let foo = let x = error "ERR" in \y -> y
let foo' = let !x = error "ERR" in \y -> y
let fooOpt = \y -> y
let foo'Opt = \y -> y -- ERROR! forcing foo' should give "ERR"
let tuple = (42, error "urk")
let bar = let (x, y) = tuple in \z -> x + z
let barOpt = \z -> 42 + z

1. bangs remove freedom from the compiler
2. bangs force large objects to be evaluated
3. Haskell is awesome because equational reasoning.
4. Bangs are not great with equational reasoning.
5. judiciously use bangs. Too many remove choice from the compiler.

Why strictness may be bad

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

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

let tuple = (42, error "urk")
let bar = let (x, y) = tuple in \z -> x + z

let barOpt = \z -> 42 + z

let bar' = let (x, !y) = tuple in \z -> x + z
```


Why strictness may be bad

let foo = let x = error "ERR" in \y -> y
let foo' = let !x = error "ERR" in \y -> y
let fooOpt = \y -> y
let foo'Opt = \y -> y -- ERROR! forcing foo' should give "ERR"
let tuple = (42, error "urk")
let bar = let (x, y) = tuple in \z -> x + z
let barOpt = \z -> 42 + z
let bar' = let (x, !y) = tuple in \z -> x + z

1. bangs remove freedom from the compiler
2. bangs force large objects to be evaluated
3. Haskell is awesome because equational reasoning.
4. Bangs are not great with equational reasoning.
5. judiciously use bangs. Too many remove choice from the compiler.

Why strictness may be bad

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

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

let tuple = (42, error "urk")
let bar = let (x, y) = tuple in \z -> x + z

let barOpt = \z -> 42 + z

let bar' = let (x, !y) = tuple in \z -> x + z

let bar'Opt =  \z -> 42 + z  --  ERROR: forcing bar' should give "urk"
```


Why strictness may be bad

1. bangs remove freedom from the compiler
2. bangs force large objects to be evaluated
3. Haskell is awesome because equational reasoning.
4. Bangs are not great with equational reasoning.
5. judiciously use bangs. Too many remove choice from the compiler.

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)
```


1. Bangs force evaluation.
2. The computation might diverge.
3. Thus the compiler can no longer move the computation around or simplify it.
4. Also if the work doesn't have to get done it still might be.
5. 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. Sometimes (point out 'intersect') we have to rearrange the code though when we use bangs.
2. Bangs tell the compiler to make more efficient code, but take away the compiler's options in how to do so.
3. Only take away the compiler's liberties when it's using them poorly.
4. After a little trial and error or reading core how to use bangs in your code will be 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. This is a far more performance critical version of what we saw with 'maximum' vs. 'max'.
2. innermost functions are of critical importance. remove Maybe which significantly reduces the boxing
3. Since a Ray that fails to intersect something can be said to intersect at infinity, Double already actually covers the structure at play
4. 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
...
```


1. Sadly 'intersects' remains one of our most costly functions even after that improvement so we'll focus on it a bit.
2. A common optimization is loop unrolling.
3. Many compilers do this for us, and there are special versions of it like Duff's Device.
4. Sadly GHC is the one compiler I've used that doesn't.
5. We can though implement every variety of it I know of by hand in Haskell, and this is one of the simplest.
6. I believe we could produce a rule that did it for a given function but I've not actually explored that.

Custom datatype for intersects parameter passing

```
Old: Tuple with possibly-unevaluated 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. Still trying to focus on the performance of 'intersects' we can optimize how it passes data to itself and its caller.
2. We want the data to be strict so the compiler knows the tuple's members will always be evaluated, but we want to avoid copying.
3. A normal tuple lacks strictness information.
4. An unboxed tuple sadly forces copying because of its unpacked nature. Sphere is quite large, and will be expensive to copy.
5. What we need is a strict tuple, and we create one for just this usage.
6. 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)
```


1. We're getting quite a bit faster now
2. there's still some small pieces of low lying fruit.
3. Strings are fairly inefficient and the conversion functions to them we use aren't particularly efficient.
4. 'bytestring' has some efficient writing code, so we just convert to that for a modest gain.

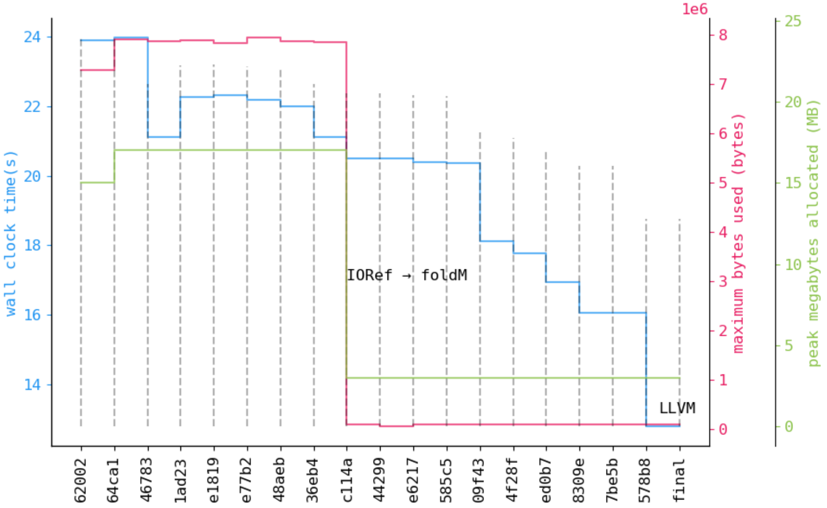
Use LLVM backend (1.87×)

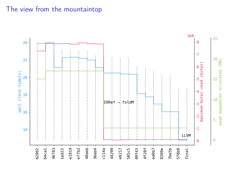
```
+package smallpt-opt
+  ghc-options: -fllvm
```


Use LLVM backend (1.87×)
+package smallpt-opt
+ ghc-options: -fllvm

1. Finally, this particular code is quite numeric heavy.
2. There are optimizations for numeric heavy code we're missing in GHC.
3. LLVM has an extensive library of laws to optimize low level numeric ops.
4. LLVM is too low-level to understand haskell as haskell.
5. 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.
6. In this case, as the code is "fortran-like", LLVM wins.

The view from the mountaintop





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. We used +Inf to match the Maybeness
2. C++ code set 1e20 s the horizon
3. Mechanical sympathy is important.
4. Know how the CPU (abstractly) executes.

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

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.

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.