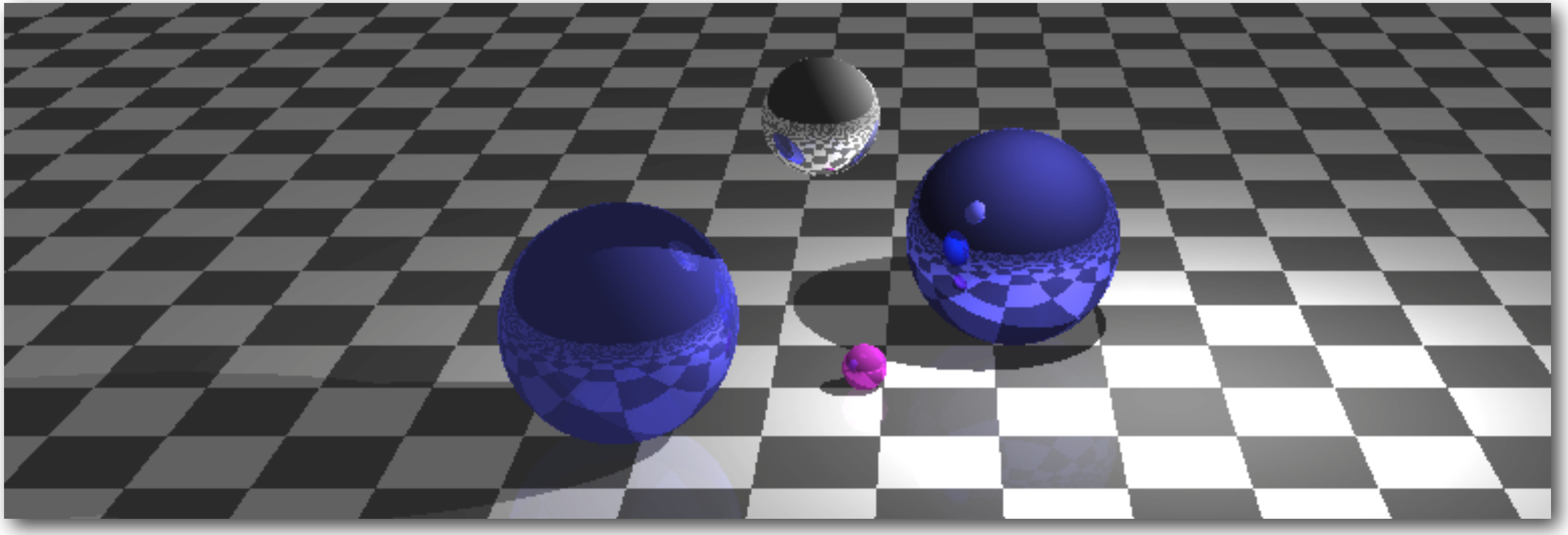
Practical Parallel Array Fusion with Repa (Workshop)

Ben Lippmeier
University of New South Wales
LambdaJam 2013

Who has...

- Written a Haskell program?
- Written a Haskell program > 1000 lines?
- Worked on a Haskell program > 10k lines?
- Uploaded a library to Hackage?
- Written Haskell code for money?
- Seen a GHC heap profile?
- Used Repa?



Real-time Parallel Ray Tracing in Haskell
(for a simple scene)

Final Ray Tracer Demos

- Show final animated ray tracer demo running.
  This is the end product.

- Show final ray tracer single image.

  ```
  $ cabal build
  $ time dist/build/ray/ray -bmp 800 600 out.bmp
  ```

  about 390 ms for a 800x600 frame, single threaded.
  about 120 ms for a 400x300 frame, single threaded.

- Show scaling with increasing number of cores.

  ```
  $ time ./Main -bmp 800 600 out.bmp +RTS -N2 -qa -qg
  ```

  Final version scales almost linearly, as we would expect.

- `+RTS -qa` : turn on thread affinity
  `+RTS -qg` : turn off parallel GC in gen 0

Naive Ray Tracer Demos

- Show original naive version, single frame.

```
$ ghc -fforce-recomp -isrc -o Main --make src/Main.hs
      -rtsopts -threaded
$ time ./Main -bmp 800 600 out.bmp
```

- Show scaling with increasing number of cores.

```
$ time ./Main -bmp 800 600 out.bmp +RTS -N2
```

  About 30 times slower, but also scales well!

- This is the #1 trap for parallel functional programmers. Haskell programs that rely on array fusion have a
  ***very high dynamic range of performance*.**

- Good speedup does **NOT** mean good performance.

Ray-tracer code walkthrough

Recap of fusion mechanism

Recap of fusion mechanism

Delayed arrays are functions!

```
data D
instance Source D e where
 data Array D sh e
  = ADelayed !sh (sh -> a)
```

Unboxed arrays are real data!

```
data U
instance Unbox e => Source U e where
 data Array U sh e
  = AUnboxed !sh (U.Vector e)
```

Recap of fusion mechanism

- Repa-style fusion with delayed arrays is critically dependent on inlining and program transformation for performance.

- With C programming, if the optimiser does not run the program is maybe 2-4 times slower.

- For Repa code, the program can be 20-40x slower.

- **Problem:** maybe the optimiser ran but could not optimise your program. How do you know what *should* have happened?

```
example :: Array D DIM2 Int
example
 = map f (zipWith g arr1 arr2)
```

```
example :: Array D DIM2 Int
example
 = map f (zipWith g arr1 arr2)
```

```
example :: Array D DIM2 Int
example
 = map f (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix -> g (arr1 !! ix) (arr2 !! ix)))
```

```
example :: Array D DIM2 Int
example
 = map f (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix -> g (arr1 !! ix) (arr2 !! ix)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ =
       g’  =
   in  map f (ADelayed (intersectDim (extent arr1) (extent arr2))
                       (\ix -> g (arr1 !! ix) (arr2 !! ix)))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  map f (ADelayed (                                        )
                       (                                   ))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  map f (ADelayed sh‘ g’)
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  map f (ADelayed sh‘ g’)
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (extent (ADelayed sh’ g’))
                (\ix2 -> f (ADelayed sh’ g’ !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (extent (ADelayed sh’ g’))
                (\ix2 -> f (ADelayed sh’ g’ !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (extent (ADelayed sh’ g’))
                (\ix2 -> f (ADelayed sh’ g’ !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (ADelayed sh’ g’ !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (ADelayed sh’ g’ !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh'
               (\ix2 -> f (ADelayed sh' g' !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 =     ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

Array Filling

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```
computeP arr
 = ...
   ...


 where
    fill !lix !end
    | lix >= end      = return ()
    | otherwise
    = do write lix
          (arr `index` fromLinearIndex lix)
         fill (lix + 1) end
    ...
```

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```
computeP (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix2 -> (arr1 !! ix2) * (arr2 !! ix2) + 1))
 = ...
   ...


 where
    fill !lix !end
    | lix >= end      = return ()
    | otherwise
    = do write lix
          (arr `index` fromLinearIndex lix)
         fill (lix + 1) end
    ...
```

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```
computeP (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix2 -> (arr1 !! ix2) * (arr2 !! ix2) + 1))
 = ...
   ...


 where
    fill !lix !end
    | lix >= end          = return ()
    | otherwise
    = do write lix
          (arr `index` fromLinearIndex lix)
         fill (lix + 1) end
    ...
```

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```
computeP (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix2 -> (arr1 !! ix2) * (arr2 !! ix2) + 1))
 = ...
   ...


 where
    fill !lix !end
    | lix >= end          = return ()
    | otherwise
    = do write lix
          (let ix′ = fromLinearIndex lix
           in  (arr1 !! ix′) * (arr2 !! ix′) + 1)
         fill (lix + 1) end
    ...
```

Glasgow Haskell Compilation Pipeline

Glasgow Haskell Compilation Pipeline

1. Lexer and Parser	(TextFile -> Haskell AST)
2. Type check and desugar	(Haskell AST -> GHC Core)
3. **Simplifier**	**(GHC Core -> GHC Core)**
4. STG Code Generation	(GHC Core -> STG language)
5. Cmm Code Generation	(STG language -> Cmm)
6. Back-end code generation	(Cmm -> LLVM)
7. Optimise and Assemble	(LLVM -> Object Code)

The GHC Simplifier

- Simplifier performs all inlining and most code transformation.
- There are other Core to Core optimisation stages that run interleaved with the simplifier: Worker Wrapper, CSE etc.
- Sometimes all optimisations passes are just referred to as “The GHC Simplifier”, though this isn’t strictly true.
- GHC Core language is designed specifically to be easy to transform and type check.
- All simplifications are correctness preserving*

* eta-expansion sometimes makes a program more terminating. see docs for `-fpedantic-bottoms`.

The GHC Core language

```
data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type     Type
  | Coercion Coercion
```

- Types and coercions can only be used as the argument of an application. For example:

```
App exp1 (Type t1)
App exp1 (Coercion t1)
```

Extracting Core Code

Extracting GHC Core code

```
$ ghc -fforce-recomp -isrc --make src/Main.hs -o Main
      -v -ddump-prep > dump.prep
```

- I almost always look at just the output of `-ddump-prep`

- This is the code just before conversion to STG.

Almost too much useful information...

```
Object.castRay
  :: [Object.Object]
     -> Vec3.Vec3
     -> Vec3.Vec3
     -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
[GblId, Arity=3, Unf=OtherCon []]
Object.castRay =
  \ (objs_s2Xk :: [Object.Object])
    (orig_s2WR :: Vec3.Vec3)
    (dir_s2WS  :: Vec3.Vec3) ->
    letrec {
      go1_s2X4 [Occ=LoopBreaker]
        :: [Object.Object]
           -> Object.Object
           -> GHC.Types.Float
           -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
      [LclId, Arity=3, Unf=OtherCon []]
      go1_s2X4 =
        \ (ds_s2WO :: [Object.Object])
          (objClose_s2WQ :: Object.Object)
          (dist_s2WT :: GHC.Types.Float) ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX :: Vec3.Vec3
```

Almost too much useful information...

```
Object.castRay
  :: [Object.Object]
     -> Vec3.Vec3
     -> Vec3.Vec3
     -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
[GblId, Arity=3, Unf=OtherCon []]
Object.castRay =
  \ (objs_s2Xk :: [Object.Object])
    (orig_s2WR :: Vec3.Vec3)
    (dir_s2WS  :: Vec3.Vec3) ->
    letrec {
      go1_s2X4 [Occ=LoopBreaker]
        :: [Object.Object]
           -> Object.Object
           -> GHC.Types.Float
           -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
      [LclId, Arity=3, Unf=OtherCon []]
      go1_s2X4 =
        \ (ds_s2WO :: [Object.Object])
          (objClose_s2WQ :: Object.Object)
          (dist_s2WT :: GHC.Types.Float) ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX :: Vec3.Vec3
```

Repeated type annots.

Almost too much useful information...

```
Object.castRay
  :: [Object.Object]
     -> Vec3.Vec3
     -> Vec3.Vec3
     -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
[GblId, Arity=3, Unf=OtherCon []]
Object.castRay =
  \ (objs_s2Xk :: [Object.Object])
    (orig_s2WR :: Vec3.Vec3)
    (dir_s2WS  :: Vec3.Vec3) ->
    letrec {
      go1_s2X4 [Occ=LoopBreaker]
        :: [Object.Object]
           -> Object.Object
           -> GHC.Types.Float
           -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
      [LclId, Arity=3, Unf=OtherCon []]
      go1_s2X4 =
        \ (ds_s2WO :: [Object.Object])
          (objClose_s2WQ :: Object.Object)
          (dist_s2WT :: GHC.Types.Float) ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX :: Vec3.Vec3
```

Explicit module prefixes.

Almost too much useful information...

```
Object.castRay
  :: [Object.Object]
     -> Vec3.Vec3
     -> Vec3.Vec3
     -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
[GblId, Arity=3, Unf=OtherCon []]
Object.castRay =
  \ (objs_s2Xk :: [Object.Object])
    (orig_s2WR :: Vec3.Vec3)
    (dir_s2WS  :: Vec3.Vec3) ->
    letrec {
      go1_s2X4 [Occ=LoopBreaker]
        :: [Object.Object]
           -> Object.Object
           -> GHC.Types.Float
           -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
      [LclId, Arity=3, Unf=OtherCon []]
      go1_s2X4 =
        \ (ds_s2WO :: [Object.Object])
          (objClose_s2WQ :: Object.Object)
          (dist_s2WT :: GHC.Types.Float) ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX :: Vec3.Vec3
```

Binding meta-data

Almost too much useful information...

```
Object.castRay
  :: [Object.Object]
     -> Vec3.Vec3
     -> Vec3.Vec3
     -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
[GblId, Arity=3, Unf=OtherCon []]
Object.castRay =
  \ (objs_s2Xk :: [Object.Object])
    (orig_s2WR :: Vec3.Vec3)
    (dir_s2WS  :: Vec3.Vec3) ->
    letrec {
      go1_s2X4 [Occ=LoopBreaker]
        :: [Object.Object]
           -> Object.Object
           -> GHC.Types.Float
           -> Data.Maybe.Maybe (Object.Object, Vec3.Vec3)
      [LclId, Arity=3, Unf=OtherCon []]
      go1_s2X4 =
        \ (ds_s2WO :: [Object.Object])
          (objClose_s2WQ :: Object.Object)
          (dist_s2WT :: GHC.Types.Float) ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX :: Vec3.Vec3
```

Unique Ids

Suppression flags

```
-dsuppress-uniques
-dsuppress-module-prefixes
-dsuppress-coercions
-dsuppress-all
```

```
$ ghc -fforce-recomp -isrc --make src/Main.hs -o Main
      -v -ddump-prep -dsuppress-all > dump.prep
```

With -dsuppress-all

```
castRay
castRay =
  \ objs_s2Xk orig_s2WR dir_s2WS ->
    letrec {
      go1_s2X4
      go1_s2X4 =
        \ ds_s2WO objClose_s2WQ dist_s2WT ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX
                sat_s2WX =
                  let {
                    sat_s2WV
                    sat_s2WV = mulsV3 dir_s2WS dist_s2WT } in
                  + $fNum(,,) orig_s2WR sat_s2WV } in
              let {
                sat_s2Zb
                sat_s2Zb = (objClose_s2WQ, sat_s2WX) } in
              Just sat_s2Zb;
            : obj_s2X1 rest_s2X3 ->
              case distanceToObject_r2BN obj_s2X1 orig_s2WR dir_s2WS of _ {
                Nothing -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
                Just dist'_s2X6 ->
                  case < $fOrdFloat dist'_s2X6 dist_s2WT of _ {
                    False -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
                    True -> go1_s2X4 rest_s2X3 obj_s2X1 dist'_s2X6
```

Problem #1: Lack of Inlining

With -dsuppress-all

```
castRay
castRay =
  \ objs_s2Xk orig_s2WR dir_s2WS ->
    letrec {
      go1_s2X4
      go1_s2X4 =
        \ ds_s2WO objClose_s2WQ dist_s2WT ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX
                sat_s2WX =
                  let {
                    sat_s2WV
                    sat_s2WV = mulsV3 dir_s2WS dist_s2WT } in
                  + $fNum(,,) orig_s2WR sat_s2WV } in
              let {
                sat_s2Zb
                sat_s2Zb = (objClose_s2WQ, sat_s2WX) } in
              Just sat_s2Zb;
            : obj_s2X1 rest_s2X3 ->
              case distanceToObject_r2BN obj_s2X1 orig_s2WR dir_s2WS of _ {
                Nothing -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
                Just dist'_s2X6 ->
                  case < $fOrdFloat dist'_s2X6 dist_s2WT of _ {
                    False -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
                    True -> go1_s2X4 rest_s2X3 obj_s2X1 dist'_s2X6
```

Problem #1: Lack of Inlining

```
castRay
castRay =
 \ objs_s2Xk orig_s2WR dir_s2WS ->
  letrec {
   go1_s2X4
   go1_s2X4 =
    \ ds_s2WO objClose_s2WQ dist_s2WT ->
     case ds_s2WO of _ {
      [] ->
       let {
        sat_s2WX
        sat_s2WX =
         let {
          sat_s2WV
          sat_s2WV = mulsV3 dir_s2WS dist_s2WT } in
         + $fNum(,,) orig_s2WR sat_s2WV } in
       let {
        sat_s2Zb
        sat_s2Zb = (objClose_s2WQ, sat_s2WX) } in
       Just sat_s2Zb;
      : obj_s2X1 rest_s2X3 ->
       case distanceToObject_r2BN obj_s2X1 orig_s2WR dir_s2WS of _ {
        Nothing -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
        Just dist'_s2X6 ->
         case < $fOrdFloat dist'_s2X6 dist_s2WT of _ {
          False -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
          True -> go1_s2X4 rest_s2X3 obj_s2X1 dist'_s2X6
```

```
mulsV3 :: Vec3 -> Float -> Vec3
mulsV3 (x1, y1, z1) s
 = (s * x1, s * y1, s * z1)
```

numeric function not inlined

Problem #1: Lack of Inlining

type class dictionary not inlined

```
castRay
castRay =
  \ objs_s2Xk orig_s2WR dir_s2WS ->
    letrec {
      go1_s2X4
      go1_s2X4 =
        \ ds_s2WO objClose_s2WQ dist_s2WT ->
          case ds_s2WO of _ {
            [] ->
              let {
                sat_s2WX
                sat_s2WX =
                  let {
                    sat_s2WV
                    sat_s2WV = mulsV3 dir_s2WS dist_s2WT } in
                  + $fNum(,,) orig_s2WR sat_s2WV } in
              let {
                sat_s2Zb
                sat_s2Zb = (objClose_s2WQ, sat_s2WX) } in
              Just sat_s2Zb;
            : obj_s2X1 rest_s2X3 ->
              case distanceToObject_r2BN obj_s2X1 orig_s2WR dir_s2WS of _ {
                Nothing -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
                Just dist'_s2X6 ->
                  case < $fOrdFloat dist'_s2X6 dist_s2WT of _ {
                    False -> go1_s2X4 rest_s2X3 objClose_s2WQ dist_s2WT;
                    True -> go1_s2X4 rest_s2X3 obj_s2X1 dist'_s2X6
```

Problem #1: Lack of Inlining

- Lack of inlining kills fusion and performance.

- All numeric functions must be inlined,
  otherwise we are paying function-call overheads for primitive operations like “add” and “multiply”

- Given standard optimisation flags,
  GHC uses heuristics to decide what to inline.

- Relying on the heuristics is fine for general Haskell programs, but not good enough if we *need* array fusion.

At least turn on inlining with heuristics

```
$ ghc -fforce-recomp -isrc --make src/Main.hs -o Main
      -v -ddump-prep -dsuppress-all -O2 > dump.prep
```

```
$ time ./Main -bmp 800 600 out.bmp
```

- Now 11 times faster than without any inlining or fusion.
- We sometimes need to add extra INLINE pragmas.
  We'll come back to this.

What performance should we expect?

- It's not always obvious how fast a program *should* be.
- Given our program is now 11 times faster... is that good?
- The fact that enabling fusion made it faster does not imply that the result will be competitive with other implementations.
- Do a rough calculation to see if we're in the ballpark.

800 x 600 image = 480k pixels

$$\frac{2.6 \text{ x } 10^9 \text{ (cycles/s) * 1 s}}{\text{480k pixels}} = \text{5400 cycles/pixel}$$

(seems high but not tragically so)

What performance should we expect?

- It's not always obvious how fast a program *should* be.
- Given our program is now 11 times faster... is that good?
- The fact that enabling fusion made it faster does not imply that the result will be competitive with other implementations.
- Do a rough calculation to see if we're in the ballpark.

800 x 600 image = 480k pixels

$$\frac{2.6 \times 10^{9} \text{ (cycles/s)} * \mathbf{10} \text{ s}}{480\text{k pixels}} = \mathbf{54000} \text{ cycles/pixel}$$

(**completely broken**)

Heap profile summary

```
./Main -bmp 800 600 out.bmp +RTS -s
```

What do you suppose it did with that 1.67 Gigs of heap?

```
 1,671,254,576 bytes allocated in the heap
     1,184,528 bytes copied during GC
     1,447,784 bytes maximum residency (3 sample(s))
       669,744 bytes maximum slop
             7 MB total memory in use (0 MB lost due to fragmentation)

                                  Tot time (elapsed)  Avg pause  Max pause
Gen  0      3259 colls,     0 par    0.01s    0.02s     0.0000s    0.0000s
Gen  1         3 colls,     0 par    0.00s    0.00s     0.0002s    0.0002s

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    1.12s  (  1.13s elapsed)
GC      time    0.02s  (  0.02s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    1.15s  (  1.15s elapsed)

%GC     time       1.3%  (1.6% elapsed)

Alloc rate    1,493,852,162 bytes per MUT second

Productivity  98.7% of total user, 98.7% of total elapsed
```

Productivity > 85% usually ok

Heap profile summary

```
./Main -bmp 800 600 out.bmp +RTS -s
```

Only ~800x600x3 bytes were ever live....

```
 1,671,254,576 bytes allocated in the heap
     1,184,528 bytes copied during GC
     1,447,784 bytes maximum residency (3 sample(s))
       669,744 bytes maximum slop
             7 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
Gen  0      3259 colls,     0 par    0.01s    0.02s     0.0000s    0.0000s
Gen  1         3 colls,     0 par    0.00s    0.00s     0.0002s    0.0002s

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    1.12s  (  1.13s elapsed)
GC      time    0.02s  (  0.02s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    1.15s  (  1.15s elapsed)

%GC     time       1.3%  (1.6% elapsed)

Alloc rate    1,493,852,162 bytes per MUT second

Productivity  98.7% of total user, 98.7% of total elapsed
```

Productivity > 85% usually ok

Extended heap profile is very suspicious....

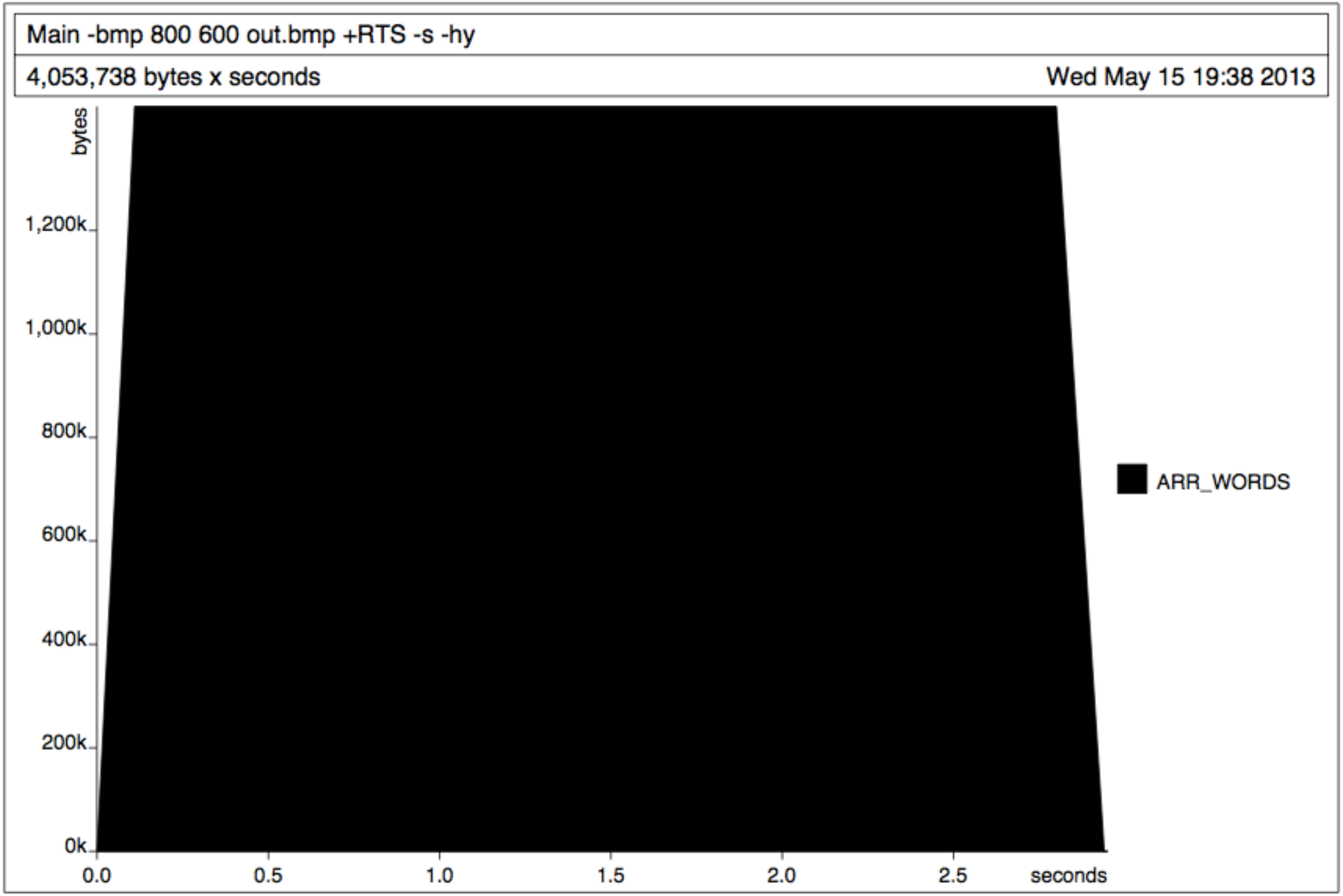
```
$ ghc -fforce-recomp -isrc --make src/Main.hs -o Main
      -v -O2 -rtsopts -prof
```

Compile with profiling.

```
$ ./Main -bmp 800 600 out.bmp +RTS -s -hy
```

Heap object type profiling.

```
$ hp2ps -c Main.ps
```



The only thing in the heap is a single ARR_WORDS??

Heap profiling continued...

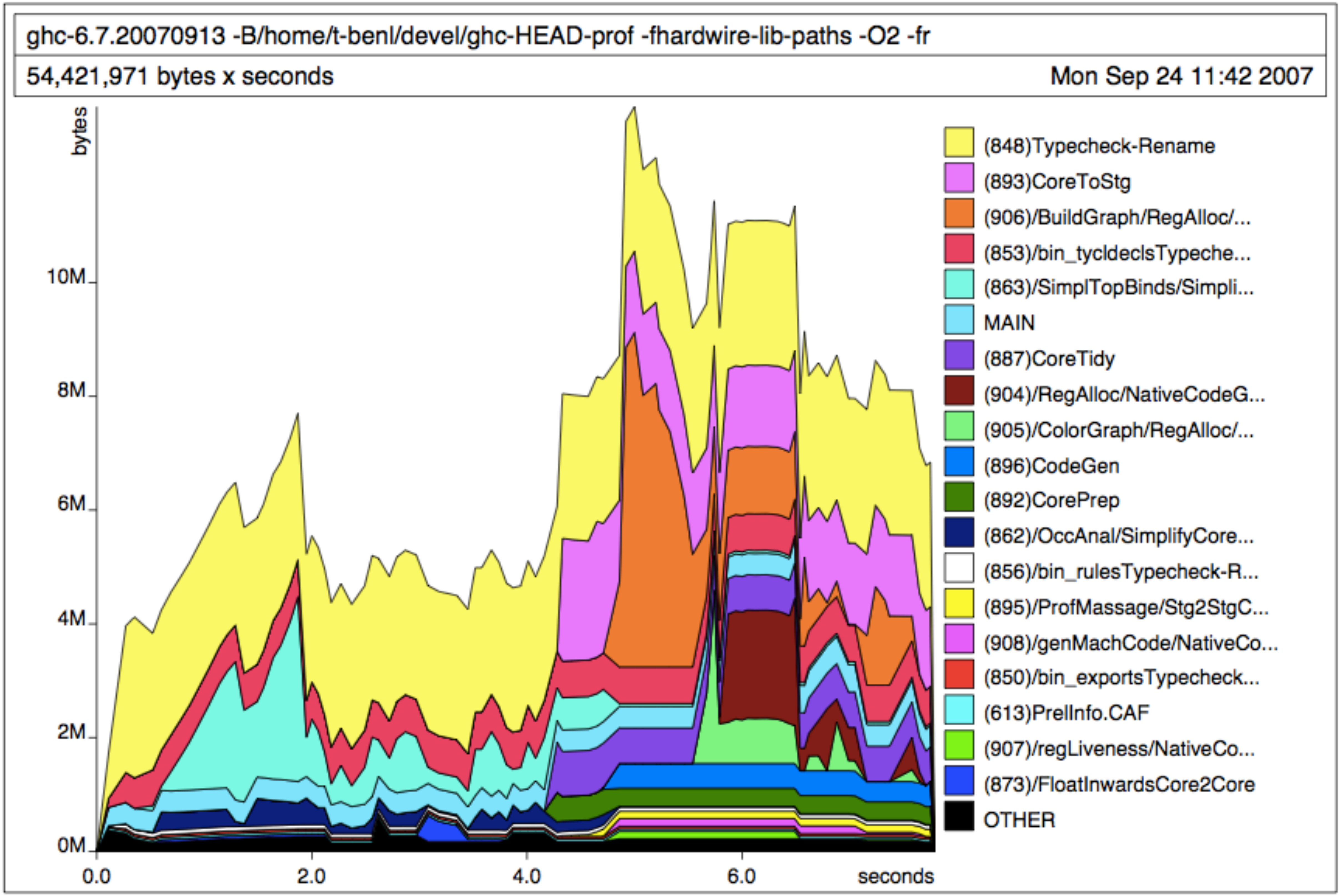
- The heap profiler reports gives the breakdown of *live* data at defined moments in time.

- If some heap allocated data died immediately then it won't show up in the heap profile.

- The GHC runtime does not provide a breakdown of what heap object types were allocated, though it will tell you the total allocation for each function.

- If your heap profile ever looks like this then assume most of the time spent in your program is boxing and then immediately unboxing numeric values.

(aside: a more interesting heap profile)



Problem #2: Boxing and Laziness

Code with heuristic inlining

```
castRay
castRay =
  \ objs_s7aO orig_s75S dir_s75X ->
    let {
      go1_s7aN
      go1_s7aN =
        \ ds_s75P objClose_s75R dist_s765 ->
          case ds_s75P of _ {
            [] ->
              let {
                sat_s76N
                sat_s76N =
                  case orig_s75S of _ { (ww_s762, ww1_s76i, ww2_s76x) ->
                  case dir_s75X of _ { (ww3_s768, ww4_s76n, ww5_s76C) ->
                  let {
                    sat_s7dt
                    sat_s7dt =
                      case ww2_s76x of _ { F# x_s76F ->
                      case dist_s765 of _ { F# x1_s76G ->
                      case ww5_s76C of _ { F# y_s76H ->
                      case timesFloat# x1_s76G y_s76H of sat_s76J { __DEFAULT ->
                      case plusFloat# x_s76F sat_s76J of sat_s7dq { __DEFAULT ->
                      F# sat_s7dq
                      }
                      }
                      }
                      }
                      }
                      } in
```

Code with heuristic inlining

```
castRay
castRay =
  \ objs_s7aO orig_s75S dir_s75X ->
    let {
      go1_s7aN
      go1_s7aN =
        \ ds_s75P objClose_s75R dist_s765 ->
          case ds_s75P of _ {
            [] ->
              let {
                sat_s76N
                sat_s76N =
                  case orig_s75S of _ { (ww_s762, ww1_s76i, ww2_s76x) ->
                  case dir_s75X of _ { (ww3_s768, ww4_s76n, ww5_s76C) ->
                  let {
                    sat_s7dt
                    sat_s7dt =
                      case ww2_s76x of _ { F# x_s76F ->
                      case dist_s765 of _ { F# x1_s76G ->
                      case ww5_s76C of _ { F# y_s76H ->
                      case timesFloat# x1_s76G y_s76H of sat_s76J { __DEFAULT ->
                      case plusFloat# x_s76F sat_s76J of sat_s7dq { __DEFAULT ->
                      F# sat_s7dq
                      }
                      }
                      }
                      }
                      }
                      } in
```

lowercase#

Primops compile into single machine instructions. These are your friends.

Code with heuristic inlining

```
castRay
castRay =
  \ objs_s7aO orig_s75S dir_s75X ->
    let {
      go1_s7aN
      go1_s7aN =
        \ ds_s75P objClose_s75R dist_s765 ->
          case ds_s75P of _ {
            [] ->
              let {
                sat_s76N
                sat_s76N =
                  case orig_s75S of _ { (ww_s762, ww1_s76i, ww2_s76x) ->
                  case dir_s75X of _ { (ww3_s768, ww4_s76n, ww5_s76C) ->
                  let {
                    sat_s7dt
                    sat_s7dt =
                      case ww2_s76x of _ { F# x_s76F ->
                      case dist_s765 of _ { F# x1_s76G ->
                      case ww5_s76C of _ { F# y_s76H ->
                      case timesFloat# x1_s76G y_s76H of sat_s76J { __DEFAULT ->
                      case plusFloat# x_s76F sat_s76J of sat_s7dq { __DEFAULT ->
                      F# sat_s7dq
                      }
                      }
                      }
                      }
                      }
                      } in
```

Uppercase#

Boxed numeric values cost heap allocation,
and correspond to lazy evaluation.
These are your enemies ~ 20 cycles each.

Code with heuristic inlining

```
castRay
castRay =
  \ objs_s7aO orig_s75S dir_s75X ->
    let {
      go1_s7aN
      go1_s7aN =
        \ ds_s75P objClose_s75R dist_s765 ->
          case ds_s75P of _ {
            [] ->
              let {
                sat_s76N
                sat_s76N =
                  case orig_s75S of _ { (ww_s762, ww1_s76i, ww2_s76x) ->
                  case dir_s75X of _ { (ww3_s768, ww4_s76n, ww5_s76C) ->
                  let {
                    sat_s7dt
                    sat_s7dt =
                      case ww2_s76x of _ { F# x_s76F ->
                      case dist_s765 of _ { F# x1_s76G ->
                      case ww5_s76C of _ { F# y_s76H ->
                      case timesFloat# x1_s76G y_s76H of sat_s76J { __DEFAULT ->
                      case plusFloat# x_s76F sat_s76J of sat_s7dq { __DEFAULT ->
                      F# sat_s7dq
                      }
                      }
                      }
                      }
                      }
                      } in
```

(,) Tuple constructors

Same as boxed values.

Also your enemies (in camouflage)

Code with heuristic inlining

```
castRay
castRay =
 \ objs_s7aO orig_s75S dir_s75X ->
  let {
   go1_s7aN
   go1_s7aN =
    \ ds_s75P objClose_s75R dist_s765 ->
     case ds_s75P of _ {
      [] ->
       let {
        sat_s76N
        sat_s76N =
         case orig_s75S of _ { (ww_s762, ww1_s76i, ww2_s76x) ->
         case dir_s75X of _ { (ww3_s768, ww4_s76n, ww5_s76C) ->
         let {
          sat_s7dt
          sat_s7dt =
           case ww2_s76x of _ { F# x_s76F ->
           case dist_s765 of _ { F# x1_s76G ->
           case ww5_s76C of _ { F# y_s76H ->
           case timesFloat# x1_s76G y_s76H of sat_s76J { __DEFAULT ->
           case plusFloat# x_s76F sat_s76J of sat_s7dq { __DEFAULT ->
           F# sat_s7dq
           }
           }
           }
           }
           }
           } in
```

non-recursive let bindings

Allocate thunks.

Also your enemies

Lazy let bindings vs “Strict-let bindings”

- Non recursive let bindings allocate thunks.
  Laziness is baked into the semantics of the core language.

```
let x = exp1 in exp2
```

- Single alternative case expressions force thunks,
  and perform primitive evaluation.
  They are called “strict let-expressions”

```
case exp1 of x { _ -> exp2 }
```

- Having lots of strict let-expressions in core dumps makes them hard to read.

```
castRay
castRay =
  \ objs_s7aO orig_s75S dir_s75X ->
    let {
      go1_s7aN
      go1_s7aN =
        \ ds_s75P objClose_s75R dist_s765 ->
          case ds_s75P of _ {
            [] ->
              let {
                sat_s76N
                sat_s76N =
                  case orig_s75S of _ { (ww_s762, ww1_s76i, ww2_s76x) ->
                  case dir_s75X of _ { (ww3_s768, ww4_s76n, ww5_s76C) ->
                  let {
                    sat_s7dt
                    sat_s7dt =
                      case ww2_s76x of _ { F# x_s76F ->
                      case dist_s765 of _ { F# x1_s76G ->
                      case ww5_s76C of _ { F# y_s76H ->
                      case timesFloat# x1_s76G y_s76H of sat_s76J { __DEFAULT ->
                      case plusFloat# x_s76F sat_s76J of sat_s7dq { __DEFAULT ->
                      F# sat_s7dq
                      }
                      }
                      }
                      }
                      }
                      } in
```

Lazy let bindings vs “Strict-let bindings”

- Use `-dppr-case-as-let` to render “strict let expressions” with a more let-expressiony syntax.

before

```
case exp1 of x { _ -> exp2 }
```

after

```
let x <- exp1 in exp2
```

```
$ ghc -fforce-recomp -isrc --make src/Main.hs -o Main
      -v -O2 -ddump-prep -dsuppress-all
      -dppr-case-as-let -dppr-cols200 > dump.prep
```

```
castRay
castRay =
 \ objs_s7aO orig_s75S dir_s75X ->
  let {
   go1_s7aN
   go1_s7aN =
    \ ds_s75P objClose_s75R dist_s765 ->
     case ds_s75P of _ {
      [] ->
       let {
        sat_s76N
        sat_s76N =
         let { (ww_s762, ww1_s76i, ww2_s76x) ~ _ <- orig_s75S } in
         let { (ww3_s768, ww4_s76n, ww5_s76C) ~ _ <- dir_s75X } in
         let {
          sat_s7dt
          sat_s7dt =
           let { F# x_s76F ~ _ <- ww2_s76x } in
           let { F# x1_s76G ~ _ <- dist_s765 } in
           let { F# y_s76H ~ _ <- ww5_s76C } in
           let { __DEFAULT ~ sat_s76J <- timesFloat# x1_s76G y_s76H } in
           let { __DEFAULT ~ sat_s7dq <- plusFloat# x_s76F sat_s76J }
           in F# sat_s7dq } in
         let {
          sat_s7du
          sat_s7du =
           let { F# x_s76q ~ _ <- ww1_s76i } in
           let { F# x1_s76r ~ _ <- dist_s765 } in
           let { F# y_s76s ~ _ <- ww4_s76n } in
           let { __DEFAULT ~ sat_s76u <- timesFloat# x1_s76r y_s76s } in
           let { __DEFAULT ~ sat_s7dr <- plusFloat# x_s76q sat_s76u }
           in F# sat_s7dr } in
```

The cost of boxing

- TRICK: To discover what assembly code a piece of source Haskell maps to, add a dummy constant to a numeric expression and compile with `-keep-s-files -fllvm`

- Add +6666 to `Object.hs:distanceToObject`

```
ghc -fforce-recomp -isrc --make src/Main.hs -o Main
    -v -O2 -ddump-prep -dsuppress-all
    -dppr-case-as-let  -dppr-cols120
    -fllvm -keep-s-files -optlo-O3 > dump.prep
```

```
-- | Compute the distance to the surface of this shape
distanceToObject
        :: Object       -- ^ Towards this object.
        -> Vec3         -- ^ Start from this point.
        -> Vec3         -- ^ Along this ray.
        -> Maybe Float  -- ^ Distance to intersection, if there is one.

distanceToObject obj orig dir
 = case obj of
    Sphere pos radius _ _
     -> let p       = orig + dir `mulsV3` ((pos - orig) `dotV3` dir)
            d_cp    = magnitudeV3 (p - pos)
        in  if    d_cp >= radius                  then Nothing
            else if (p - orig) `dotV3` dir <= 0.0 then Nothing
            else Just $ magnitudeV3 (p - orig) + 66666
                      - sqrt (radius * radius - d_cp * d_cp)

    Plane pos normal _ _
     -> if dotV3 dir normal >= 0.0
           then Nothing
           else Just (((pos - orig) `dotV3` normal) / (dir `dotV3` normal))

    PlaneCheck pos normal _
     -> if dotV3 dir normal >= 0.0
           then Nothing
           else Just (((pos - orig) `dotV3` normal) / (dir `dotV3` normal))
```

This is all numeric code.
At assembly level we might hope for a few branches
and the rest primitive arithmetic.

```
$wdistanceToObject
$wdistanceToObject =
  \ w_scLz w1_scLF ww_scMe ww1_scMj ww2_scMq ->
    case w_scLz of _ {
      Sphere pos_scLK radius_scM7 ds_sd2W ds1_sd2X ->
        let { (ww3_scLP, ww4_scLY, ww5_scM4) ~ _ <- w1_scLF } in
        let { (ww6_scLS, ww7_scLV, ww8_scM1) ~ _ <- pos_scLK } in
        let { F# x_scMc ~ _ <- ww3_scLP } in
        let { F# x1_scMb ~ _ <- ww6_scLS } in
        let { F# x2_scMg ~ _ <- ww7_scLV } in
        let { F# y_scMh ~ _ <- ww4_scLY } in
        let { F# x3_scMn ~ _ <- ww8_scM1 } in
        let { F# y1_scMo ~ _ <- ww5_scM4 } in
        let { F# y2_scMX ~ _ <- radius_scM7 } in
        let { __DEFAULT ~ sat_scYL <- minusFloat# x3_scMn y1_scMo } in
        let { __DEFAULT ~ sat_scMu <- timesFloat# sat_scYL ww2_scMq } in
        let { __DEFAULT ~ sat_scYK <- minusFloat# x2_scMg y_scMh } in
        let { __DEFAULT ~ sat_scMl <- timesFloat# sat_scYK ww1_scMj } in
        let { __DEFAULT ~ sat_scYJ <- minusFloat# x1_scMb x_scMc } in
        let { __DEFAULT ~ sat_scYI <- timesFloat# sat_scYJ ww_scMe } in
        let { __DEFAULT ~ sat_scMv <- plusFloat# sat_scYI sat_scMl } in
        let { __DEFAULT ~ x4_scMs <- plusFloat# sat_scMv sat_scMu } in
        let { __DEFAULT ~ sat_scMz <- timesFloat# x4_scMs ww2_scMq } in
        let { __DEFAULT ~ x5_scMx <- plusFloat# y1_scMo sat_scMz } in
        let { __DEFAULT ~ x6_scMA <- minusFloat# x5_scMx x3_scMn } in
        let { __DEFAULT ~ sat_scMF <- timesFloat# x4_scMs ww1_scMj } in
        let { __DEFAULT ~ x7_scMD <- plusFloat# y_scMh sat_scMF } in
        let { __DEFAULT ~ x8_scMG <- minusFloat# x7_scMD x2_scMg } in
```

We arrive at numeric primops **(good)** after lots of tedious unboxing **(bad)**.

```
_Object_zdwdistanceToObject_info_itable:
    .quad    _Object_zdwdistanceToObject_slow-
_Object_zdwdistanceToObject_info
    .quad    1797                      ## 0x705
    .quad    0                         ## 0x0
    .quad    21474836480               ## 0x500000000
    .quad    0                         ## 0x0
    .quad    15                        ## 0xf
    .text
    .globl   _Object_zdwdistanceToObject_info
    .align   3, 0x90
_Object_zdwdistanceToObject_info:         ## @Object_zdwdistanceToObject_info
## BB#0:                                  ## %cmEY
    leaq-80(%rbp), %rax
    cmpq%r15, %rax
    jae  LBB280_1
## BB#3:
    movq    %r14, -40(%rbp)
    movq    %rsi, -32(%rbp)
    movss   %xmm1, -24(%rbp)
    movss   %xmm2, -16(%rbp)
    movss   %xmm3, -8(%rbp)
    movq   -8(%r13), %rax
    addq   $-40, %rbp
    leaq   _Object_zdwdistanceToObject_closure(%rip), %rbx
    jmpq* %rax   # TAILCALL
LBB280_1:                                 ## %nmF6
    movss   %xmm3, -32(%rbp)
    movss   %xmm2, -24(%rbp)
    movss   %xmm1, -16(%rbp)
    movq    %rsi, -8(%rbp)
    leaq_sd2V_info(%rip), %rax
    movq   %rax, -40(%rbp)
    ....
```

info table

Copy args to stack and indirect jump.
The signature of lazy evaluation.
Look at how much of the assembly
code just does this....

“entry code”

```
LCPI244_0:
    .long   1199715584              ## float 6.666600e+04

....
    addss   %xmm12, %xmm3
    subss   %xmm10, %xmm0
    mulss   %xmm0, %xmm7
    addss   %xmm3, %xmm7
    xorps   %xmm3, %xmm3
    ucomiss %xmm7, %xmm3
    jae LBB244_5
## BB#3:                                   ## %nlve
    mulss   %xmm6, %xmm6
    mulss   %xmm2, %xmm2
    addss   %xmm6, %xmm2
    mulss   %xmm0, %xmm0
    addss   %xmm2, %xmm0
    mulss   %xmm4, %xmm4
    mulss   %xmm1, %xmm1
    subss   %xmm4, %xmm1
    sqrtss  %xmm1, %xmm1
    sqrtss  %xmm0, %xmm0
    movq_ghczmprim_GHCziTypes_Fzh_con_info@GOTPCREL(%rip), %rcx
    movq   %rcx, 8(%r12)
    leaq   -6(%rax), %rbx
    leaq   -23(%rax), %rcx
    movq_base_DataziMaybe_Just_con_info@GOTPCREL(%rip), %rdx
    addss   LCPI244_0(%rip), %xmm0
    subss   %xmm1, %xmm0
    movss   %xmm0, 16(%r12)
```

```
LCPI244_0:
    .long   1199715584              ## float 6.666600e+04

....
    addss   %xmm12, %xmm3
    subss   %xmm10, %xmm0
    mulss   %xmm0, %xmm7
    addss   %xmm3, %xmm7
    xorps   %xmm3, %xmm3
    ucomiss %xmm7, %xmm3
    jae LBB244_5
## BB#3:                                    ## %nlve
    mulss   %xmm6, %xmm6
    mulss   %xmm2, %xmm2
    addss   %xmm6, %xmm2
    mulss   %xmm0, %xmm0
    addss   %xmm2, %xmm0
    mulss   %xmm4, %xmm4
    mulss   %xmm1, %xmm1
    subss   %xmm4, %xmm1
    sqrtss  %xmm1, %xmm1
    sqrtss  %xmm0, %xmm0
    movq_ghczmprim_GHCziTypes_Fzh_con_info@GOTPCREL(%rip), %rcx
    movq    %rcx, 8(%r12)
    leaq    -6(%rax), %rbx
    leaq    -23(%rax), %rcx
    movq_base_DataziMaybe_Just_con_info@GOTPCREL(%rip), %rdx
    addss   LCPI244_0(%rip), %xmm0
    subss   %xmm1, %xmm0
    movss   %xmm0, 16(%r12)
```

Real arithmetic instructions here.
This is the “real computation”.
Most of the rest is junk due to boxing and laziness.

Change #1: Use strict unboxed data

- Do not use the tuple type in data structures.

- Put bang patterns on all data structure fields.

- Use `-funpack-strict-fields` to tell GHC to store values of types like `Int` and `Float` directly in structures with no boxing overhead and no laziness.

before

```
type Vec3
    = (Float, Float, Float)

data Object
    = Sphere
    { spherePos       :: Vec3
    , sphereRadius    :: Float
    , sphereColor     :: Color
    , sphereShine     :: Float }
```

after

```
data Vec3
   = Vec3 !Float !Float !Float

data Object
    = Sphere
    { spherePos       :: !Vec3
    , sphereRadius    :: !Float
    , sphereColor     :: !Color
    , sphereShine     :: !Float }
```

before

```
type Vec3
    = (Float, Float, Float)

data Object
    = Sphere
    { spherePos       :: Vec3
    , sphereRadius    :: Float
    , sphereColor     :: Color
    , sphereShine     :: Float }
```

after

```
data Vec3
   = Vec3 !Float !Float !Float

data Object
    = Sphere
    { spherePos       :: !Vec3
    , sphereRadius    :: !Float
    , sphereColor     :: !Color
    , sphereShine     :: !Float }
```

With `-funbox-strict-fields` the three `Float` components will be unboxed and unpacked into the runtime Object structure.

before

```
$ /usr/bin/time ./Main -bmp 800 600 out.bmp
```

1133 ms / frame

after

```
$ /usr/bin/time ./Main -bmp 800 600 out.bmp
```

470 ms / frame

now 2x faster...

```
ghc -fforce-recomp -isrc --make src/Main.hs -o Main
    -v -O2 -funbox-strict-fields
    -ddump-prep -dsuppress-all
    -dppr-case-as-let  -dppr-cols120
    -fllvm -keep-s-files -optlo-O3 > dump.prep
```

before

```
$wdistanceToObject
$wdistanceToObject =
  \ w_scLz w1_scLF ww_scMe ww1_scMj ww2_scMq ->
    case w_scLz of _ {
      Sphere pos_scLK radius_scM7 ds_sd2W ds1_sd2X ->
        let { (ww3_scLP, ww4_scLY, ww5_scM4) ~ _ <- w1_scLF } in
        let { (ww6_scLS, ww7_scLV, ww8_scM1) ~ _ <- pos_scLK } in
        let { F# x_scMc ~ _ <- ww3_scLP } in
        let { F# x1_scMb ~ _ <- ww6_scLS } in
        let { F# x2_scMg ~ _ <- ww7_scLV } in
        let { F# y_scMh ~ _ <- ww4_scLY } in
```

after

No more unboxing of single floats..

```
$wdistanceToObject
$wdistanceToObject =
  \ w_sdjE w1_sdjO ww_sdjX ww1_sdk2 ww2_sdk9 ->
    case w_sdjE of _ {
      Sphere rb_sdjU rb1_sdjZ rb2_sdk6 rb3_sdkG rb4_sdxb rb5_sdxc rb6_sdxd rb7_sdxe ->
        let { Vec3 rb8_sdjV rb9_sdk0 rb10_sdk7 ~ _ <- w1_sdjO } in
        let { __DEFAULT ~ sat_sdtH <- minusFloat# rb2_sdk6 rb10_sdk7 } in
        let { __DEFAULT ~ sat_sdkd <- timesFloat# sat_sdtH ww2_sdk9 } in
        let { __DEFAULT ~ sat_sdtG <- minusFloat# rb1_sdjZ rb9_sdk0 } in
        let { __DEFAULT ~ sat_sdk4 <- timesFloat# sat_sdtG ww1_sdk2 } in
```

```
ghc -fforce-recomp -isrc --make src/Main.hs -o Main
    -v -O2 -funbox-strict-fields
    -ddump-prep -dsuppress-all
    -dppr-case-as-let  -dppr-cols120
    -fllvm -keep-s-files -optlo-O3 > dump.prep
```

before

```
$wdistanceToObject
$wdistanceToObject =
  \ w_scLz w1_scLF ww_scMe ww1_scMj ww2_scMq ->
    case w_scLz of _ {
      Sphere pos_scLK radius_scM7 ds_sd2W ds1_sd2X ->
        let { (ww3_scLP, ww4_scLY, ww5_scM4) ~ _ <- w1_scLF } in
        let { (ww6_scLS, ww7_scLV, ww8_scM1) ~ _ <- pos_scLK } in
        let { F# x_scMc ~ _ <- ww3_scLP } in
        let { F# x1_scMb ~ _ <- ww6_scLS } in
        let { F# x2_scMg ~ _ <- ww7_scLV } in
        let { F# y_scMh ~ _ <- ww4_scLY } in
```

... but a boxed vector is being passed as a parameter and then unboxed....

after

```
$wdistanceToObject
$wdistanceToObject =
  \ w_sdjE w1_sdjO ww_sdjX ww1_sdk2 ww2_sdk9 ->
    case w_sdjE of _ {
      Sphere rb_sdjU rb1_sdjZ rb2_sdk6 rb3_sdkG rb4_sdxb rb5_sdxc rb6_sdxd rb7_sdxe ->
        let { Vec3 rb8_sdjV rb9_sdk0 rb10_sdk7 ~ _ <- w1_sdjO } in
        let { __DEFAULT ~ sat_sdtH <- minusFloat# rb2_sdk6 rb10_sdk7 } in
        let { __DEFAULT ~ sat_sdkd <- timesFloat# sat_sdtH ww2_sdk9 } in
        let { __DEFAULT ~ sat_sdtG <- minusFloat# rb1_sdjZ rb9_sdk0 } in
        let { __DEFAULT ~ sat_sdk4 <- timesFloat# sat_sdtG ww1_sdk2 } in
```

Problem #2.5: *oh no, not more* Boxing and Laziness.

You really need to kill all boxing and unboxing.

- .. at least in inner loops.
- ... it costs to much ...
- Trawl through the core code looking for it.

```
castRay
castRay =
  \ objs_sdrz orig_sdo5 dir_sdoa ->
    let {
      go1_sdry =
        \ ds_sdo2 objClose_sdo4 dist_sdof ->
          case ds_sdo2 of _ {
            [] ->
              let {
                sat_sdoz =
                  let { Vec3 rb_sdoi rb1_sdoo rb2_sdot  ~ _ <- orig_sdo5 } in
                  let { Vec3 rb3_sdok rb4_sdop rb5_sdou ~ _ <- dir_sdoa } in
                  let { F# x_sdoj ~ _ <- dist_sdof } in
                  let { __DEFAULT ~ sat_sdow <- timesFloat# x_sdoj rb5_sdou } in
                  let { __DEFAULT ~ sat_sdu3 <- plusFloat# rb2_sdot sat_sdow } in
                  let { __DEFAULT ~ sat_sdor <- timesFloat# x_sdoj rb4_sdop } in
                  let { __DEFAULT ~ sat_sdu4 <- plusFloat# rb1_sdoo sat_sdor } in
                  let { __DEFAULT ~ sat_sdom <- timesFloat# x_sdoj rb3_sdok } in
                  let { __DEFAULT ~ sat_sdu5 <- plusFloat# rb_sdoi sat_sdom } in
                  Vec3 sat_sdu5 sat_sdu4 sat_sdu3 } in
```

Remember: non-recursive let expressions create thunks.

This one is created because the result is packed into a (non-strict) tuple.

```
castRay
castRay =
  \ objs_sdrz orig_sdo5 dir_sdoa ->
    let {
      go1_sdry =
        \ ds_sdo2 objClose_sdo4 dist_sdof ->
          case ds_sdo2 of _ {
            [] ->
              let {
                sat_sdoz =
                  let { Vec3 rb_sdoi rb1_sdoo rb2_sdot  ~ _ <- orig_sdo5 } in
                  let { Vec3 rb3_sdok rb4_sdop rb5_sdou ~ _ <- dir_sdoa } in
                  let { F# x_sdoj ~ _ <- dist_sdof } in
                  let { __DEFAULT ~ sat_sdow <- timesFloat# x_sdoj rb5_sdou } in
                  let { __DEFAULT ~ sat_sdu3 <- plusFloat# rb2_sdot sat_sdow } in
                  let { __DEFAULT ~ sat_sdor <- timesFloat# x_sdoj rb4_sdop } in
                  let { __DEFAULT ~ sat_sdu4 <- plusFloat# rb1_sdoo sat_sdor } in
                  let { __DEFAULT ~ sat_sdom <- timesFloat# x_sdoj rb3_sdok } in
                  let { __DEFAULT ~ sat_sdu5 <- plusFloat# rb_sdoi sat_sdom } in
                  Vec3 sat_sdu5 sat_sdu4 sat_sdu3 } in
```

Remember: non-recursive let expressions create thunks.

This one is created because the result is packed into a (non-strict) tuple.

```
castRay objs orig dir
 = go0 objs
 where -- We hit an object before, and we're testing others
       -- to see if they're closer.
       go1 [] objClose dist
        = Just (objClose, orig + dir `mulsV3` dist)
```

Change #2: Strictify all non-function binders.

- Add bang patterns on all non-function binders.
(putting them on function binders can prevent inlining)

- Add bang patterns on all let-bindings.

- Use `` `seq` `` to strictify any left-over tuple components.

Bang patterns on parameters and accumulators

before

```
castRay objs orig dir
 = go0 objs
 where -- We hit an object before, and we're testing others
       -- to see if they're closer.
       go1 [] objClose dist
        = Just (objClose, orig + dir `mulsV3` dist)
```

after

```
castRay objs !orig !dir
 = go0 objs
 where -- We hit an object before, and we're testing others
       -- to see if they're closer.
       go1 [] objClose !dist
        = Just (objClose, orig + dir `mulsV3` dist)
```

Add bang patterns to let bindings

before

```
let   -- Size of the raw image to render.
      sizeX = winSizeX `div` zoomX
      sizeY = winSizeY `div` zoomY
in   ...
```

after

```
let   -- Size of the raw image to render.
      !sizeX = winSizeX `div` zoomX
      !sizeY = winSizeY `div` zoomY
in   ...
```

Use `seq` to strictify any left over tuple components

before

```
playField !display (zoomX, zoomY) !stepRate
          !initWorld makePixel handleEvent stepWorld
 = if zoomX < 1 || zoomX < 1
    then ...
```

after

```
playField !display (zoomX, zoomY) !stepRate
          !initWorld makePixel handleEvent stepWorld
 = zoomX `seq` zoomY `seq`
   if zoomX < 1 || zoomX < 1
    then ...
```

``x `seq` y`` evaluate `x` to whnf and yield `y`

What about strictness analysis?

- The strictness analyser can (usually) determine when a variable is used strictly.

- We want more strictness than the default semantics provide.

- Even if you think strictness analysis will recover the information, add the strictness annotations anyway.

- It’s easier to scan source code looking for missing annotations than to think about whether each variable is strict.

- In high performance numeric code you almost never want lazy evaluation.

before

```
$ /usr/bin/time ./Main -bmp 800 600 out.bmp
```

470 ms / frame

after

```
$ /usr/bin/time ./Main -bmp 800 600 out.bmp
```

360 ms / frame

about 30% faster.

Ok, now what?

```
$wtraceRay
$wtraceRay =
  \ w_saEf w1_saEh ww_saGe ww1_saGC ww2_saGq ww3_saL4 ww4_saL5
    ww5_saL6 w2_saEj ww6_saL3 ->
    let { __DEFAULT ~ objs_saEy <- w_saEf } in
    let { __DEFAULT ~ lights_saG2 <- w1_saEh } in
    let { Vec3 ipv_saFA ipv1_saFC ipv2_saFG ~ _ <- w2_saEj } in
    letrec {
      $s$wgo_saFM
      $s$wgo_saFM =
        \ sc_saEw sc1_saEz sc2_saEA sc3_saEB sc4_saEC sc5_saED sc6_saEE ->
          case sc_saEw of wild_saFL {
            __DEFAULT ->
              case $wcastRay objs_saEy sc1_saEz sc2_saEA sc3_saEB
                             sc4_saEC sc5_saED sc6_saEE of _ {
                Nothing -> (# __float 0.0, __float 0.0, __float 0.0 #);
                Just ds_saEH ->
                  let { (obj_saEW, point_saEL) ~ _ <- ds_saEH } in
                  let { Vec3 rb_saFf rb1_saFb rb2_saF7 ~ _ <- point_saEL } in
                  let { $w$j_saKz $w$j_saKz =
                      \ w3_saG8 ->
                        let {
                          $w$j1_saJx
                          $w$j1_saJx = ...
```

castRay is returning a boxed object which must then be unboxed.

(worker)

```
$wcastRay
$wcastRay =
  \ w_s7n0 ww_s7dd ww1_s7dj ww2_s7do ww3_s7df ww4_s7dk ww5_s7dp ->
    letrec {
      $sgo1_s7dG
      $sgo1_s7dG =
        \ sc_s7da sc1_s7dc sc2_s7de ->
          case sc_s7da of _ {
            [] ->
              let { __DEFAULT ~ sat_s7dr <- timesFloat# sc2_s7de ww5_s7dp } in
              let { __DEFAULT ~ sat_s7dt <- plusFloat# ww2_s7do sat_s7dr } in
```

(wrapper)

```
castRay
castRay =
  \ w_s7ne w1_s7n4 w2_s7n9 ->
    let { Vec3 ww_s7nf ww1_s7ng ww2_s7nh ~ _ <- w1_s7n4 } in
    let { Vec3 ww3_s7ni ww4_s7nj ww5_s7nk ~ _ <- w2_s7n9 } in
    $wcastRay w_s7ne ww_s7nf ww1_s7ng ww2_s7nh ww3_s7ni ww4_s7nj ww5_s7nk
```

- The GHC worker wrapper transform can often cause parameters to be unboxed, but doesn’t help so much with results.

```
case $wcastRay objs_saEy sc1_saEz sc2_saEA sc3_saEB
                             sc4_saEC sc5_saED sc6_saEE of _ {
                Nothing -> (# __float 0.0, __float 0.0, __float 0.0 #);
                Just ds_saEH ->
                  let { (obj_saEW, point_saEL) ~ _ <- ds_saEH } in
                  let { Vec3 rb_saFf rb1_saFb rb2_saF7 ~ _ <- point_saEL } in
```

- To kill this you could try forcing `castRay` to be inlined.
- Add `{-# INLINE castRay #-}` after its definition site.
- ... no such luck. It makes the program slower.

  360 -> 400 ms / frame
- ... though inlining `traceRay` and `tracePixel` seems to help.
- Too much inlining can cause instruction cache miss.

Change #3: Rewrite structure producers to use continuation passing style (CPS)

- Doing so eliminates the branch on Nothing/Just from the consumer.

before

```
castRay :: [Object]        -- check for intersections on all these objects
        -> Vec3            -- ray origin
        -> Vec3            -- ray direction
        -> Maybe
                ( Object   -- object of first intersected
                , Vec3)    -- position of intersection, on surface of object
```

360 ms / frame

after

```
-- | Like castRay, but take continuations for the Nothing and Just branches to
--   eliminate intermediate unboxings.
castRay_continuation
        :: [Object]        -- check for intersections on all these objects
        -> Vec3            -- ray origin
        -> Vec3            -- ray direction

        -> a                     -- continuation when no intersection
        -> (Object -> Vec3 -> a) -- continuation with intersection
        -> a
```

340 ms / frame