Graphics

How do we do graphical output?

Ofcourse there are bindings to all possible GUI toolkits:

- gtk, Qt, wxWidgets
- SDL, OpenGL

Haskell-specific stuff (more interesting!):

- Back-end: JuicyPixels (&repa)
- Generative graphics: Rasterific
- Plots: Chart
- Interactive graphics: Gloss & NotGloss

```
import Codec.Picture
readImage :: FilePath → IO (Either String DynamicImage)
```

DynamicImage unifies all possible pixel representations, including ImageRGBA8 (Image PixelRGBA8) or ImageYCbCr8 (Image PixelYCbCr8).

```
writeDynamicBitmap ::
  FilePath → DynamicImage → IO (Either String Bool)
writeBitmap :: BmpEncodable pixel ⇒
  FilePath → Image pixel → IO ()
writePng :: PngSavable pixel ⇒
  FilePath → Image pixel → IO ()

  ⋮
```

What to do with a Picture?

Make a picture out of a function:

$$generateImage :: \text{Pixel } px \Rightarrow (\text{Int} \to \text{Int} \to px) \to \text{Int} \to \text{Int} \to \text{Image } px$$

map-like simplification:

$$pixelMap :: \text{Pixel } a, \text{Pixel } b \Rightarrow (a \to b) \to \text{Image } a \to \text{Image } b$$

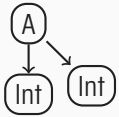
Pixels

Pixels are numbers

```
data PixelRGB8 = PixelRGB8 ! Pixel8 ! Pixel8 ! Pixel8
type PixelF = Float
data PixelCMYK16 = PixelCMYK16 ! Pixel16 ! Pixel16
                               ! Pixel16 ! Pixel16
```

The exclamation mark in the data type means that the field is not 'boxed'. Among other, there's less overhead and the field cannot be less defined than the whole structure.

```
data A = A Int Int
```




```
data A = A ! Int ! Int
```

The extra pointers hurt especially if they point to a huge amount of tiny things.

Strict&Unboxed (details)

Avoiding the unnecessary pointers can be helped a little:

$$seq :: a \rightarrow b \rightarrow b$$

“The value of seq a b is bottom if a is bottom, and otherwise equal to b. In other words, it evaluates the first argument a to weak head normal form (WHNF). seq is usually introduced to improve performance by avoiding unneeded laziness.”

```
f ! a = ...
```

```
f $! a
```

```
{-# INLINE f #-}
```

What does that look like in the memory?

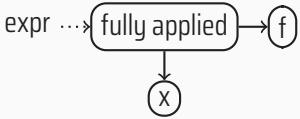
$$f :: X \to A$$
$$expr = f$$

What does that look like in the memory?

$f :: X \to A$

$expr = f$

$expr = f\ x$

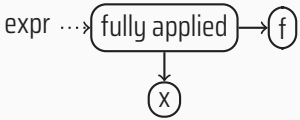


What does that look like in the memory?

$f :: X \to A$

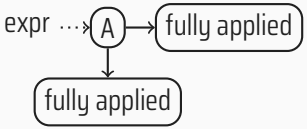
expr = *f*

expr = *f x*



data A = A Int Int

expr = *f* $! *x*

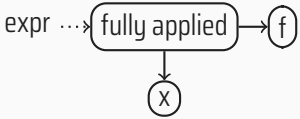


What does that look like in the memory?

$f :: X \to A$

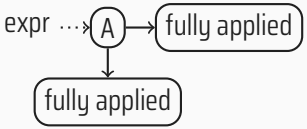
expr = *f*

expr = *f x*



data A = A Int Int

expr = *f* $! *x*



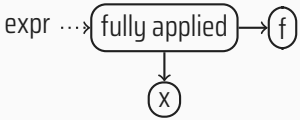
(intermediate step: *deepSeq*)

What does that look like in the memory?

$f :: X \to A$

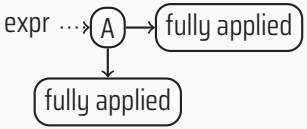
expr = *f*

expr = *f x*



data A = A Int Int

expr = *f* $! *x*



(intermediate step: *deepSeq*)

data A = A ! Int ! Int

expr = *f* $! *x*

Strict&Unboxed (practically)

Writing the exclamation marks manually is boring. Let's have these in libraries instead.

- Data.Vector.Unboxed, Data.Vector.Primitive
- Data.Set.Strict, Data.Map.Strict, …
- Control.Monad.Trans:
  - State.Strict, Writer.Strict, …
  - RWS.Strict
- *strict* :: Strict *lazy strict* ⇒ Iso' *lazy strict*
- *unpackStrict* :: ByteString → [Word8]
- *foldl'*

Repa

‘Regular parallel arrays’ (polymorphic multidimensional primitive strict arrays)

- regular ⟶ fast
- array function fusing ⟶ faster
- lowlevel optimizations when compiling with LLVM backend ⟶ fasterer
- parallel processing comes almost for free

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
...
```

- R.Array U DIM2 Float — matrix
- R.Array V Z Int — 1 point
- (Z :. 1 :. 2 :. 3) — index into a 3D array
- (Z :. 3) — index into a 1D array

```
map :: (Shape sh, Source r a) ⇒
  (a → b) → Array r sh a
   → Array D sh b
zipWith :: (Shape sh, Source r1 a, Source r2 b) ⇒
  (a → b → c) → Array r1 sh a → Array r2 sh b
   → Array D sh c
fromFunction ::
  sh → (sh → a) → Array D sh a
```

Running the computation:

```
computeUnboxedS :: (Load r1 sh e, Unbox e) ⇒
  Array r1 sh e → Array U sh e
computeUnboxedP :: (Load r1 sh e, Monad m, Unbox e) ⇒
  Array r1 sh e → m (Array U sh e)
```

```
map :: (Shape sh, Source r a) ⇒
  (a → b) → Array r sh a
   → Array D sh b
zipWith :: (Shape sh, Source r1 a, Source r2 b) ⇒
  (a → b → c) → Array r1 sh a → Array r2 sh b
   → Array D sh c
fromFunction ::
  sh → (sh → a) → Array D sh a
```

Running the computation:

```
computeUnboxedS :: (Load r1 sh e, Unbox e) ⇒
  Array r1 sh e → Array U sh e
computeUnboxedP :: (Load r1 sh e, Monad m, Unbox e) ⇒
  Array r1 sh e → m (Array U sh e)
```

Repa (usage)

```
ghc -O -fllvm -threaded -rtsopts program.hs
./program +RTS -N 16
```

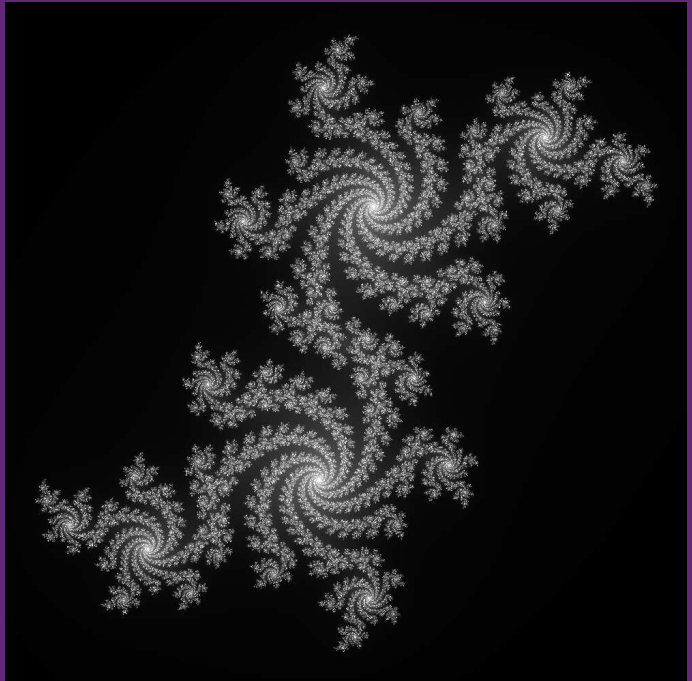
Example (back to JuicyPixels)

```
import Data.Complex
import Data.Word
import Codec.Picture

julia limit z a0 = run a0 0
  where run c i | i ⩾ limit = limit
                | magnitude c > 2 = i
                | otherwise = run (c * c + z) (i + 1)

juliapix x y = (fromIntegral :: Int → Word8) $
  julia 255 (0.141 :+ 0.6) $
  ((−1.25) :+ (−1.25)) + 0.0025 * (fromIntegral x :+ fromIntegral y)

main = writePng "fractal.png" $ generateImage juliapix 1000 1000
```



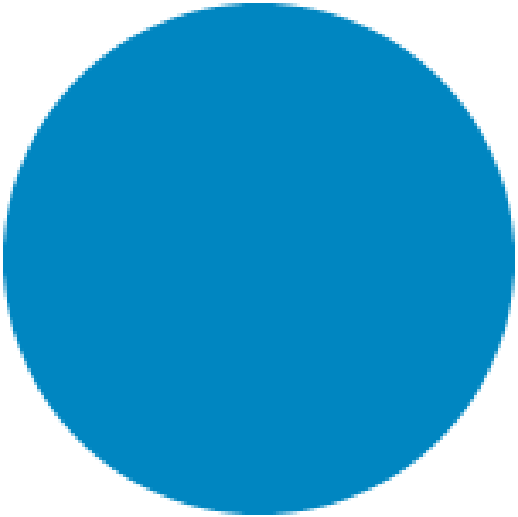
How to draw something more complex?

```
import Codec.Picture (PixelRGBA8 (..), writePng)
import Graphics.Rasterific

white = PixelRGBA8 255 255 255 255
drawColor = PixelRGBA8 0 0x86 0xc1 255

main = writePng "roundboi.png" $
  renderDrawing 200 200 white $
    fill $ circle (V2 100 100) 75
```

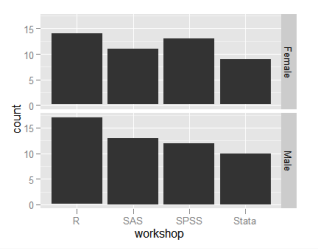
More libraries: `diagrams` (vector diagrams, alignment, arrows, ...), `juicypixels-repa` (more advanced image processing), `friday` (common raster image computations such as blur)

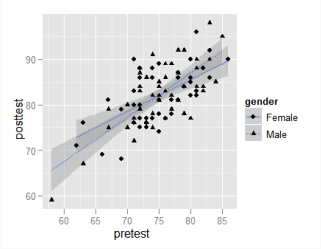


Plots

Plotting good plots is a hard task

State-of-art: `ggplot2`





GGplot:

```
plot(data) + geom_point(pretest,posttest) + geom_quantile() + title(...)
```

Haskell: Individual pieces of the plots are substructures, we can dig into them using lenses.

Chart

```
import Graphics.Rendering.Chart
import Data.Colour
import Data.Colour.Names
import Data.Default.Class
import Graphics.Rendering.Chart • Backend.Cairo
import Control.Lens

setLinesBlue :: PlotLines a b → PlotLines a b
setLinesBlue = plot_lines_style • line_color .~ opaque blue

chart = toRenderable layout
  where
    am :: Double → Double
    am x = (sin (x ∗ 3.14159 / 45) + 1) / 2 ∗ (sin (x ∗ 3.14159 / 5))

    sinusoid1 = plot_lines_values .~ [ [ (x, (am x)) | x ← [ 0, (0.5) . . 400 ] ] ]
                $ plot_lines_style • line_color .~ opaque blue
                $ plot_lines_title .~ "am" $ def

    sinusoid2 = plot_points_style .~ filledCircles 2 (opaque red)
                $ plot_points_values .~ [ (x, (am x)) | x ← [ 0, 7 . . 400 ] ]
                $ plot_points_title .~ "am points" $ def

    layout = layout_title .~ "Amplitude Modulation"
           $ layout_plots .~ [ toPlot sinusoid1, toPlot sinusoid2 ] $ def

main = renderableToFile def "example1_big.png" chart
```

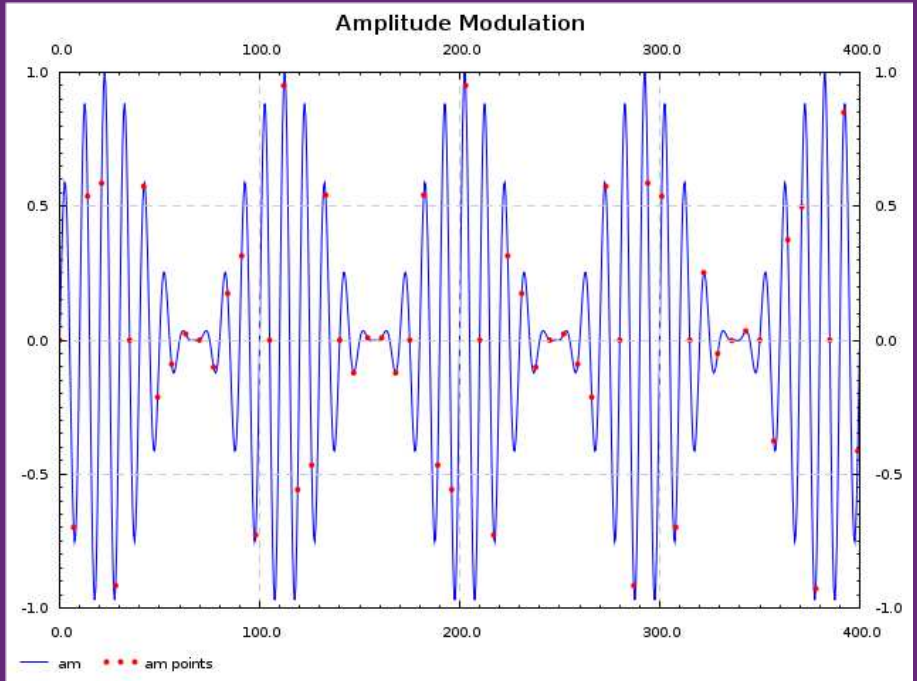

Amplitude Modulation
0.0
100.0
200.0
300.0
400.0
1.0
0.5
0.0
-0.5
-1.0
am
am points

Chart (Monads!)

```
import Graphics.Rendering.Chart.Easy
import Graphics.Rendering.Chart.Backend.Cairo

signal :: [Double] → [(Double, Double)]
signal xs = [(x, (sin (x * 3.14159 / 45) + 1)
                 / 2 * (sin (x * 3.14159 / 5)))
            | x ← xs]

main = toFile def "example1_big.png" $ do
  layout_title . = "Amplitude Modulation"
  setColors [opaque blue, opaque red]
  plot (line "am" [signal [0, (0.5) . . 400]])
  plot (points "am points" (signal [0, 7 . . 400]))
```

“Get something cool on the screen in under 10 minutes.”

Gloss

Gloss gives you a simple data type for describing the scene, and quite efficient rendering via OpenGL.

Scene:

```
type Point = (Float, Float)
type Path = [Point]
data Picture =
    | Blank
    | Polygon Path | Line Path | Circle Float
    | Text String
    | Color Color Picture
    | Translate Float Float Picture
    | Rotate Float Picture | Scale Float Picture
    | ...
```

Is it fast?

Is it fast?

It is lazy. The “scene” never gets materialized.

```
import Graphics.Gloss
display :: Display → Color → Picture → IO ()
main = display (InWindow "Window!" (200, 200) (10, 10))
  white (ThickCircle 50 3)

animate :: Display → Color → (Float → Picture) → IO ()
main = animate FullScreen
  white $ λt → Pictures $ flip map [1..4] $ λx →
  Rotate (90 * x + 20 * t) $
  Translate 80 0 $
  ThickCircle 20 (3 + 2 * sin t)
```

```
simulate :: Display
         → Color
         → Int          - fps
         → model        - anything
         → (model → Picture)
         → (ViewPort → Float → model → model)
         → IO ()
```

```
play :: Display
     → Color
     → Int    - fps
     → world    - world description
     → (world → Picture)    - render
     → (Event → world → world)
     → (Float → world → world)
     → IO ()
```

3D: not-gloss

```
data VisObject a = VisObjects [VisObject a]
    | Trans (V3 a) (VisObject a)
    | RotQuat (Quaternion a) (VisObject a)
    | RotEulerDeg (Euler a) (VisObject a)
    | Scale (a, a, a) (VisObject a)
    | Cylinder (a, a) GlossColor.Color
    | Box (a, a, a) Flavour GlossColor.Color
    | Ellipsoid (a, a, a) Flavour GlossColor.Color
    | Line (Maybe a) [V3 a] GlossColor.Color
    | Arrow (a, a) (V3 a) GlossColor.Color
    | Plane (V3 a) GlossColor.Color GlossColor.Color
    | Triangle (V3 a) (V3 a) (V3 a) GlossColor.Color
    | Quad (V3 a) (V3 a) (V3 a) (V3 a) GlossColor.Color
    | Text3d String (V3 a) BitmapFont GlossColor.Color
    | Text2d String (a, a) BitmapFont GlossColor.Color
    | Points [V3 a] (Maybe GLfloat) GlossColor.Color
    | ObjModel LoadedObjModel GlossColor.Color
    | ...
```

```
play :: Real b ⇒
  Options
   → Double    - sample time
   → world     - initial state
   → (world → (VisObject b, Maybe Cursor))    - draw function
   → (Float → world → world)    - state propogation function
   → (world → IO ())    - set where camera looks
   → Maybe (world → Key → KeyState → Modifiers → Position → world)
   → Maybe (world → Position → world)    - mouse drag
   → Maybe (world → Position → world)    - mouse move
   → IO ()
```

Debugging, testing, benchmarking

How do we even debug Haskell?

Problem: Looking at the runtime evaluation and “variables” with a debugger doesn’t make much sense.

- In fact that’s good.
- Variables don’t exist very long before they are discarded
- Computation order is quite hard to digest for humans
- Structure of compiled code differs vastly from the source

How do we even debug Haskell?

Problem: Looking at the runtime evaluation and “variables” with a debugger doesn’t make much sense.

- In fact that’s good.
- Variables don’t exist very long before they are discarded
- Computation order is quite hard to digest for humans
- Structure of compiled code differs vastly from the source

Instead:

- We’ve got a good interpreter
- The type system is useful
- We can trace stuff (which is the ultimate debugging method anyway)

The program can't be compiled because of types

Common programmers' brains unfortunately don't contain the type system

Useful backup: _ a.k.a. Typed Hole

```
_ (+1) [1, 2, 3]
```

Found hole: _ :: (Integer → Integer) → [Integer] → *t*

The program can't be compiled because of types

Common programmers' brains unfortunately don't contain the type system

Useful backup: _ a.k.a. Typed Hole

```
_ (+1) [1, 2, 3]
```

Found hole: _ :: (Integer → Integer) → [Integer] → *t*

```
foldl _ 0 ["asd", "qwe"]
```

Found hole:

The program can't be compiled because of types

Common programmers' brains unfortunately don't contain the type system

Useful backup: _ a.k.a. Typed Hole

```
_ (+1) [1, 2, 3]
```

Found hole: _ :: (Integer → Integer) → [Integer] → *t*

```
foldl _ 0 ["asd", "qwe"]
```

Found hole:_ :: *b* → [Char] → *b*
Where: *b* is a rigid type variable with inferred type Num *b* ⇒ *b*

It's quite useful to just copypaste this type to Hoogle.

Type error is too complex!

As a common problem, errors propagate to different functions via inferred types.

```
let concatMonoids = foldr mempty (◇)
 ...
print $ concatMonoids ["this","that"]
```

Type error is too complex!

As a common problem, errors propagate to different functions via inferred types.

```
let concatMonoids = foldr mempty (◇)
 ...
print $ concatMonoids ["this", "that"]
```

No instance for (Show ($m0 \to m0 \to m0$)) ?

Type error is too complex!

```
let concatMonoids = foldr mempty (◇)
 ...
print $ (concatMonoids ["a", "bcd"] :: _)
```

Found type wildcard _ standing for $m0 \to m0 \to m0$

Type error is too complex!

```
let concatMonoids :: _
    concatMonoids = foldr mempty (<>)
 ...
print $ concatMonoids ["a", "bcd"]
```

Found type wildcard _ standing for $[a] \to m0 \to m0 \to m0$

...okay wait this doesn't match the expectation!

Type error is too complex!

$$\textbf{let } concatMonoids :: \text{Monoid } m \Rightarrow [m] \to m$$
$$\quad concatMonoids = foldr\ mempty\ (\diamond)$$
$$\ldots$$
$$print\ \$\ concatMonoids\ [\texttt{"a"}, \texttt{"bcd"}]$$

Couldn't match type m with $m0 \to m0 \to m0$
In the expression: $foldr\ mempty\ (\diamond)$

The error message doesn't go deeper, let's blame *foldr*.

```
:t foldr
```

$$foldr :: \text{Foldable } t \Rightarrow (a \to b \to b) \to b \to t\ a \to b$$

Type error is too complex!

With the fix applied, it's always useful to check if we've been generic enough:

```
:t foldr (<>) mempty
```

$$foldr\ (\diamond)\ mempty :: (\text{Monoid}\ b, \text{Foldable}\ t) \Rightarrow t\ b \rightarrow b$$

Trace

How do we look at what the program works with in the middle of some computation?

- C-style: `printf` anywhere!
- Haskell, SW development solution:
  Convert all functions to IO so that *print* works!!
- Haskell, a good non-invasive temporary solution: `Debug.Trace`

Trace

How do we look at what the program works with in the middle of some computation?

- C-style: `printf` anywhere!
- Haskell, SW development solution:
  Convert all functions to IO so that *print* works!!
- Haskell, a good non-invasive temporary solution: Debug.Trace

trace :: String → *a* → *a*

```
ghci> if 3 < 4 then trace "here!" 5 else 6
here!
5
```

(This is the actual side effect. Stuff can get broken; this has no place in production code!)

Trace

trace :: String → *val* → *val*
traceShow :: Show *msg* ⇒→ *msg* → *val* → *val*
traceShowId :: Show *val* ⇒ *val* → *val*

traceM :: Applicative *f* ⇒ String → *f* ()
traceShowM :: (Show *msg*, Applicative *f*) ⇒ *msg* → *f* ()

```
import Debug.SimpleReflect

product [1..10] :: Expr
  ⇝ 1*1*2*3*4*5*6*7*8*9*10

foldr (+) 0 [1..4] :: Expr
  ⇝ 1 + (2 + (3 + (4 + 0)))

map f [1,2,3] :: [Expr]
  ⇝ [f 1,f 2,f 3]

scanl f 0 [1..3] :: [Expr]
  ⇝ [0,f 0 1,f (f 0 1) 2,f (f (f 0 1) 2) 3]

mapM_ print $ reduction (1*2 + 3*4)
  ⇝ 1*2 + 3*4
    2 + 3*4
    2 + 12
    14
```

Unit tests

Unit tests aren't very popular in Haskell; sometimes even perceived as a bad habit from other languages.

- ☹ test-driven development is replaced by type-driven development
- ☹ absence of side effects makes `ghci` effective even for huge programs
- ☹ unit tests are not integration tests
- ☹ polymorphism drastically reduces the coverage volume
- ☼ good unit tests are more of a higher-class documentation
- ☼ in teams, unit tests protect you from your favorite colleagues

Unit tests

Unit tests aren't very popular in Haskell; sometimes even perceived as a bad habit from other languages.

- ☹ test-driven development is replaced by type-driven development
- ☹ absence of side effects makes `ghci` effective even for huge programs
- ☹ unit tests are not integration tests
- ☹ polymorphism drastically reduces the coverage volume
- ☼ good unit tests are more of a higher-class documentation
- ☼ in teams, unit tests protect you from your favorite colleagues

Unit tests

Unit tests aren't very popular in Haskell; sometimes even perceived as a bad habit from other languages.

- ☹ test-driven development is replaced by type-driven development
- ☹ absence of side effects makes `ghci` effective even for huge programs
- ☹ unit tests are not integration tests
- ☹ polymorphism drastically reduces the coverage volume
- ☼ good unit tests are more of a higher-class documentation
- ☼ in teams, unit tests protect you from your favorite colleagues

Unit tests

Unit tests aren't very popular in Haskell; sometimes even perceived as a bad habit from other languages.

- ☹ test-driven development is replaced by type-driven development
- ☹ absence of side effects makes `ghci` effective even for huge programs
- ☹ unit tests are not integration tests
- ☹ polymorphism drastically reduces the coverage volume
- ☼ good unit tests are more of a higher-class documentation
- ☼ in teams, unit tests protect you from your favorite colleagues

Unit tests

Unit tests aren't very popular in Haskell; sometimes even perceived as a bad habit from other languages.

- ☹ test-driven development is replaced by type-driven development
- ☹ absence of side effects makes `ghci` effective even for huge programs
- ☹ unit tests are not integration tests
- ☹ polymorphism drastically reduces the coverage volume
- ☼ good unit tests are more of a higher-class documentation
- ☼ in teams, unit tests protect you from your favorite colleagues

Unit tests

Unit tests aren't very popular in Haskell; sometimes even perceived as a bad habit from other languages.

- ☹ test-driven development is replaced by type-driven development
- ☹ absence of side effects makes `ghci` effective even for huge programs
- ☹ unit tests are not integration tests
- ☹ polymorphism drastically reduces the coverage volume
- ☼ good unit tests are more of a higher-class documentation
- ☼ in teams, unit tests protect you from your favorite colleagues

HUnit

- `cabal` can be set up to run tests
- `test/` contains a program that runs the tests
- We use `Test.HUnit` to make a nice interface for that program

`test/main.hs`

```
import FindIdentifier (findIdentifier)
import Test.HUnit

testEmpty = TestCase $ assertEqual "Should get Nothing from an empty string"
  Nothing (findIdentifier "" (1,1))

testNegCursor = TestCase $ assertEqual "Should get Nothing when cursor is negative"
  Nothing (findIdentifier "a" (-1,-1))

testMinimal = TestCase $ assertEqual "Minimal program"
  (Just "main") (findIdentifier "main = print 42" (1,2))

main = runTestTT $ TestList [testEmpty, testNegCursor, testMinimal]
```

QuickCheck

Writing tests manually is a nuisance. Let's use a fuzzer!

```
import Test.QuickCheck

prop_reverse :: [Int] → Bool
prop_reverse xs = reverse (reverse xs) ≡ xs
```

```
ghci> quickCheck prop_reverse
 >>> quickCheck prop_reverse
 +++ OK, passed 100 tests.
```

QuickCheck

```
ghci> quickCheck $ \a -> a + 1 == a + 1
+++ OK, passed 100 tests.
ghci> quickCheck $ \a -> abs (a*a) == a*a
+++ OK, passed 100 tests.

ghci> quickCheck $ \a -> abs (a*a*a) == a*a*a
*** Failed! Falsifiable (after 4 tests and 2 shrinks):
-1
```

```
ghci> quickCheck $
  \a -> (if a == 20 then 666 else a) == a
+++ OK, passed 100 tests.

ghci> quickCheck . withMaxSuccess 10000 $
  \a -> (if a == 20 then 666 else a) == a
*** Failed! Falsifiable (after 22 tests):
20
```

```
ghci> quickCheck $ \xs n -> xs !! n == head (drop n xs)
*** Failed! Exception: 'Prelude.!!: index too large'
[]
0

ghci> quickCheck $ \(NonEmpty xs) (NonNegative n) ->
        xs !! n == head (drop n xs)
*** Failed! Exception: 'Prelude.!!: index too large'
NonEmpty {getNonEmpty = [()]}
NonNegative {getNonNegative = 1}
```

QuickCheck

```
ghci> quickCheck $ \(NonEmpty xs) (NonNegative n) ->
        n < length xs ==> xs !! n == head (drop n xs)

+++ OK, passed 100 tests; 61 discarded.
```

How does QuickCheck find stuff to test?

How does QuickCheck find examples to test?

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a → [a]
```

- Gen is State-like environment that gives randomness
- *arbitrary* should be able to make a random object of type *a*
- *shrink* should convert a bigger random object to smaller sub-objects (that are used as a sub-cases for locating the actual problem)

Benchmarking

Benchmarks typically require:

- multiple runs
- multiple data sizes
- avoiding the pitfall with measuring the lazy non-evaluation

Haskell solution: Criterion + DeepSeq

Criterion — demo

Let's see how much slower Integer is to Int.

$$sumN\ n = sum\ [1\,..\,n]$$
$$fact\ n = product\ [1\,..\,n]$$

$$sumNi = sumN :: \mathrm{Int} \to \mathrm{Int}$$
$$sumNhi = sumN :: \mathrm{Integer} \to \mathrm{Integer}$$
$$facti = fact :: \mathrm{Int} \to \mathrm{Int}$$
$$facthi = fact :: \mathrm{Integer} \to \mathrm{Integer}$$

Criterion — demo

Let's see how much slower Integer is to Int.

```
sumN n = sum [1..n]
fact n = product [1..n]

sumNi = sumN :: Int → Int
sumNhi = sumN :: Integer → Integer
facti = fact :: Int → Int
facthi = fact :: Integer → Integer


problemSizes = [1, 30..100]

mkBench f size = bench (show size) $ nf f $ fromInteger size
mkBenches f = map (mkBench f) problemSizes
```

Criterion — demo

Let's see how much slower Integer is to Int.

```
sumN n = sum [1..n]
fact n = product [1..n]

sumNi = sumN :: Int → Int
sumNhi = sumN :: Integer → Integer
facti = fact :: Int → Int
facthi = fact :: Integer → Integer


problemSizes = [1, 30..100]

mkBench f size = bench (show size) $ nf f $ fromInteger size
mkBenches f = map (mkBench f) problemSizes
```

Criterion — demo

Let's see how much slower Integer is to Int.

```
sumN n = sum [1..n]
fact n = product [1..n]

sumNi = sumN :: Int → Int
sumNhi = sumN :: Integer → Integer
facti = fact :: Int → Int
facthi = fact :: Integer → Integer


problemSizes = [1, 30..100]

mkBench f size = bench (show size) $ nf f $ fromInteger size
mkBenches f = map (mkBench f) problemSizes
```

Criterion — demo

```
main = defaultMain [
  bgroup "sum-int" $ mkBenches sumNi,
  bgroup "sum-integer" $ mkBenches sumNhi,
  bgroup "fact-int" $ mkBenches facti,
  bgroup "fact-integer" $ mkBenches facthi]
```

Hint: remember to compile with `-O`

Criterion — demo

```
main = defaultMain [
  bgroup "sum-int" $ mkBenches sumNi,
  bgroup "sum-integer" $ mkBenches sumNhi,
  bgroup "fact-int" $ mkBenches facti,
  bgroup "fact-integer" $ mkBenches facthi]
```

Hint: remember to compile with `-O`

Criterion — demo

```
$ ./ints --help
Microbenchmark suite - built with criterion 1.5.2.0

Usage: ints [-I|--ci CI] [-L|--time-limit SECS]
            [--resamples COUNT] ...

$ ./ints
benchmarking sum-int/1
time                 8.832 ns   (8.723 ns .. 8.952 ns)
                     0.996 R2   (0.991 R2 .. 0.999 R2)
mean                 9.068 ns   (8.829 ns .. 9.675 ns)
std dev              1.246 ns   (536.7 ps .. 2.682 ns)
variance introduced by outliers: 96% (severely inflated)
...
```

Criterion — output

```
benchmarking sum-int/88
time                 75.10 ns   (74.46 ns .. 76.07 ns)

benchmarking sum-integer/88
time                 1.529 us   (1.516 us .. 1.550 us)

benchmarking fact-int/88
time                 103.7 ns   (99.13 ns .. 108.3 ns)

benchmarking fact-integer/88
time                 3.645 us   (3.593 us .. 3.723 us)
```

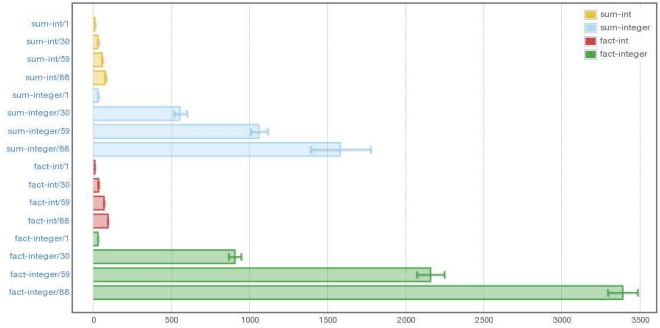
Criterion — output

```
$ ./ints --output output.html
```

criterion performance measurements

overview

want to understand this report?

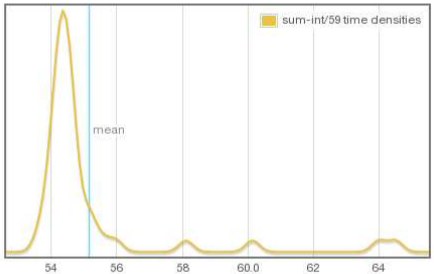


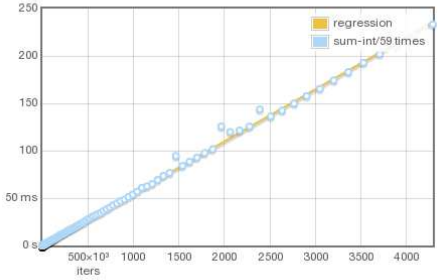
sum-int/1

Criterion — output

```
$ ./ints --output output.html
```

sum-int/59





	lower bound	estimate	upper bound
OLS regression	54.5 ns	**55.0 ns**	55.7 ns
R^2 goodness-of-fit	0.997	**0.998**	1.000
Mean execution time	54.7 ns	**55.2 ns**	56.2 ns
Standard deviation	1.22 ns	**2.30 ns**	3.69 ns

Outlying measurements have severe (63.5%) effect on estimated standard deviation.

`nf` on a custom type

Problem: generically it is not quite clear how (and how much) to force the more complex types.

```
nf :: Control.DeepSeq.NFData b ⇒
      (a → b) → a → Benchmarkable

data Tree = Leaf Int | Branch Tree Tree
instance NFData Tree where
  rnf (Leaf i)       = rnf i `seq` ()
  rnf (Branch l r) = rnf l `seq` rnf r `seq` ()
```

Definition of *seq* does not correspond to evaluation order!

`nf` on a custom type

Problem: generically it is not quite clear how (and how much) to force the more complex types.

```
nf :: Control.DeepSeq.NFData b ⇒
      (a → b) → a → Benchmarkable

data Tree = Leaf Int | Branch Tree Tree
instance NFData Tree where
  rnf (Leaf i)       = rnf i `seq` ()
  rnf (Branch l r) = rnf l `seq` rnf r `seq` ()
```

Definition of *seq* does not correspond to evaluation order!

If *a* has a weak head-normal form (WHNF), *seq a b* gives *b*. If *a* has no WHNF, *seq a b* also doesn't have WHNF.

(intuitively: *min a b*)