Writing Fast Haskell

Elegance Is Not an Excuse for Bad Performance

Moritz Kiefer (`@cocreature`)

August 22, 2018

Introduction

- Haskellers often talk about elegant code

Introduction

- Haskellers often talk about elegant code
- But elegance is *not* an excuse for bad performance!

Introduction

- Haskellers often talk about elegant code
- But elegance is *not* an excuse for bad performance!
- Writing fast Haskell requires some understanding of GHC's internals

Introduction

- Haskellers often talk about elegant code
- But elegance is *not* an excuse for bad performance!
- Writing fast Haskell requires some understanding of GHC’s internals
- GHC provides a surprising number of tools to influence performance

Goals for Today

1. Learn to reason about performance
2. Look under the hood of GHC (specifically Core)
3. Learn about some rules of thumb for writing fast Haskell
4. Learn about primitives and libraries useful for writing fast Haskell

Benchmarking/Profiling Disclaimer

- Benchmark before you optimize
- GHC supports options for time and space profiling
- Profiling can break optimizations
    - Enable profiling selectively

Primitive, Unlifted and Boxed Types

Primitive Types

Correspond to "raw machine types"

E.g. `Int#`, `Double#`

Primitive, Unlifted and Boxed Types

Primitive Types

Correspond to "raw machine types"

E.g. `Int#`, `Double#`

Boxed Types

Represented by a pointer to a heap object

E.g. all user-defined types, `Int`

Primitive, Unlifted and Boxed Types

Primitive Types

Correspond to “raw machine types”

E.g. `Int#`, `Double#`

Boxed Types

Represented by a pointer to a heap object

E.g. all user-defined types, `Int`

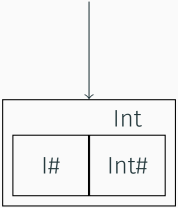
Unlifted Types

Cannot be bottom

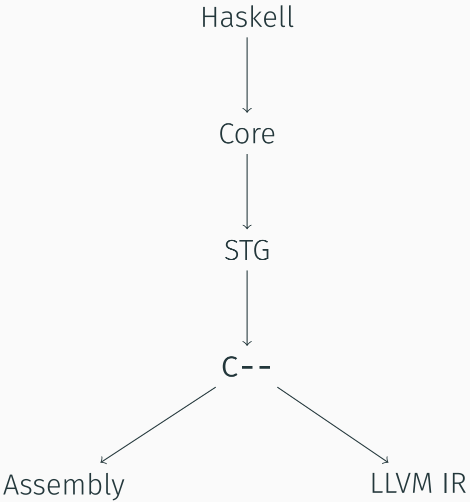
E.g. all primitive types but also `Array#` (which is not primitive)

Definition of the Int type

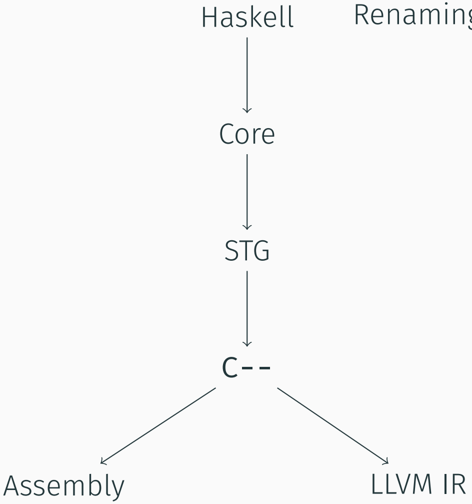
```
data Int = I# Int#
```



GHC Compilation Pipeline

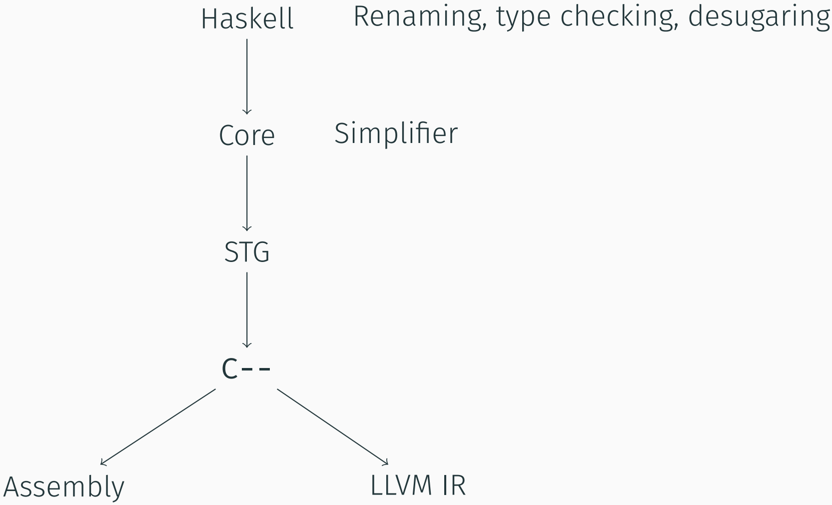


GHC Compilation Pipeline

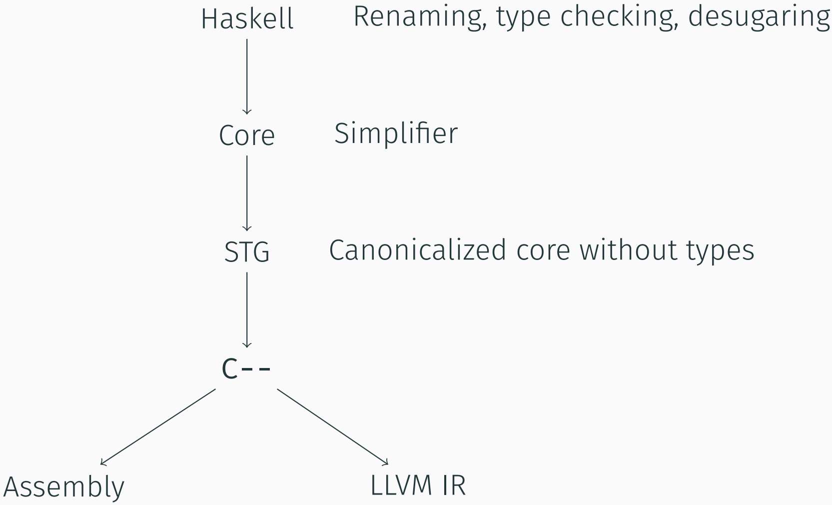


Renaming, type checking, desugaring

GHC Compilation Pipeline



GHC Compilation Pipeline



Core's Expr Type

```haskell
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
  deriving Data
```

Viewing Core

- `-ddump-simpl` or `-ddump-prep`
- Suppress info that you don't care about
    - `-dsuppress-idinfo`
    - `-dsuppress-ticks`
    - `-dsuppress-module-prefixes`
    - `-dsuppress-all`
- GHC plugin that outputs core as HTML

Mental Model for Core

`let`
Allocates a thunk on the heap

`case`
Forces evaluation to WHNF

Naive Sum

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs
```

Tail-Recursive Sum

```haskell
sum :: [Int] -> Int
sum = go 0
  where
    go acc [] = acc
    go acc (x : xs) = go (x + acc) xs
```

Force the Accumulator

```haskell
sum :: [Int] -> Int
sum = go 0
  where
    go acc [] = acc
    go acc (x : xs) =
      let acc' = x + acc
      in acc' `seq` go acc' xs
```

BangPatterns

```haskell
{-# LANGUAGE BangPatterns #-}
sum :: [Int] -> Int
sum = go 0
  where
    go acc [] = acc
    go acc (x : xs) =
      let !acc' = x + acc
      in go acc' xs
```

WHNF

- `seq` only evaluates to WHNF
- Be careful with tuples!
  `(x,y) `seq` ` … will not evaluate `x` and `y`
- Use the `deepseq` lib for evaluating to NF

Strictness Annotations in Data Types

```haskell
data Point = Point !Int !Int
```

Whenever you evaluate `Point` to WHNF, you also evaluate the two fields to WHNF.

Often easier to use than `seq`/`BangPatterns`

Avoiding Space Leaks

Rule of Thumb

Constant-size (e.g. Int) accumulators are often problematic

Detecting Space Leaks

- Limit the stack size `+RTS -K${n}K`
- Get a stacktrace with `+RTS -xc -K${n}K`

Specialization and Inlining

Specialization

- Specialize type parameters
- Remove type class dictionaries

Inlining

- Inline definition at call site

Cross-Module Specialization and Inlining

- Specialization/Inlining only possible if definition (=unfolding) is available
- Unfoldings of small definitions are automatically exposed
- `{-# INLINABLE f #-}` forces GHC to expose `f`'s unfolding
- You might also want to expose unfoldings of definitions used by `f`

Specialization

- GHC will automatically try to specialize definitions at use-sites
- Create specializations using `{-# SPECIALIZE f :: Int -> Int #-}`
  - Also creates specializations of functions called by `f`

Inlining

- `{-# INLINE f #-}` makes GHC very eager to inline `f`
- Use cautiously!
    - Can blow up compile times significantly
    - Can increase code size without bringing benefits

Inlining

- `{-# INLINE f #-}` makes GHC very eager to inline `f`
- Use cautiously!
    - Can blow up compile times significantly
    - Can increase code size without bringing benefits
- Note: `{-# INLINABLE f #-}` does *not* make GHC more eager to inline `f`

Inlining

The following two definitions are equivalent.

```
f a b = …
```

```
f = \a b …
```

Inlining

Or are they?

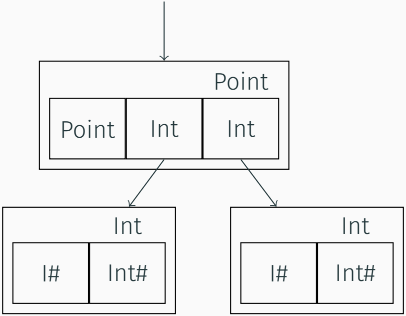
```
f a b = …

f = \a b …
```

GHC will only inline
fully saturated function applications!

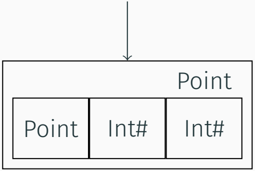
Controlling Memory Layout

```
data Point =
  Point Int
        Int
```



Controlling Memory Layout

```haskell
data Point =
  Point {-# UNPACK #-} !Int
        {-# UNPACK #-} !Int
```



Automatic Unpacking

- GHC is quite good at automatic unpacking
- But only if it can detect that an argument is strict
- Sometimes you need to help it

```haskell
f :: Vector Int -> …
f xs = …
  where n = Vector.length xs
```

Automatic Unpacking

- GHC is quite good at automatic unpacking
- But only if it can detect that an argument is strict
- Sometimes you need to help it

```
f :: Vector Int -> …
f xs = …
  where !n = Vector.length xs
```

Continuation Passing Style

```haskell
main :: IO ()
main =
  case loop2 100 (10, 10) of
    (x, y) -> print (x - y)

loop2 :: Int -> (Int,Int) -> (Int,Int)
loop2 n (x, y)
  | n > 0     = loop2 (n - 1) (x + 1, y - 1)
  | otherwise = (x, y)
```

Continuation Passing Style

Convert

```
f :: a -> b
```

into

```
f :: a -> (b -> r) -> r
```

Can avoid allocations and unnecessary case distinctions

Continuation Passing Style

```haskell
main :: IO ()
main =
  loop2 100 (10, 10) $ \(x, y) -> print (x - y)

loop2 :: Int -> (Int,Int) -> ((Int,Int) -> r) -> r
loop2 n (x, y) cont
  | n > 0     = loop2 (n - 1) (x + 1,y - 1) cont
  | otherwise = cont (x, y)
```

Writing Your Own Optimizations: Rewrite Rules

`map` Fusion

```
map f . map g = map (f . g)
```

Writing Your Own Optimizations: Rewrite Rules

map Fusion

```
map f . map g = map (f . g)
```

build/foldr Fusion

```
build
  :: (forall b. (a -> b -> b) -> b -> b)
  -> [a]
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f a (build g) = g f a
```

Writing your Own Optimizations: Rewrite Rules

Example

```
{-# RULES
"map/map"
  forall f g xs. map f (map g xs) =
                 map (f.g) xs
#-}
```

- GHC does *not* check correctness of rules
- GHC does *not* check termination of rules
- Use *phases* to control interaction of rules and inlining

Array Primitives in GHC

`Array#`

- Array of boxed values
- Card table to avoid having to scan unmodified entries in GC

`SmallArray#`

- Array of boxed values
- No card table

`ByteArray#`

- Region of raw memory
- Pinned and unpinned

High-Level Array Libraries

- `primitive` provides `PrimArray` wrapper around `ByteArray#`
- `vector` provides boxed, unboxed and `Storable` vectors
    - Fusion
    - Slicing

Basic Data Structures

- `containers` is mostly pretty good!
    - Use the specialized data structures for `Int`: `IntSet` and `IntMap`
- `unordered-containers` has a fast, persistent `HashMap`
- Mutable hashtables from the `hashtables` package are often slower

Conclusion

- GHC is impressively good at optimizing high-level code
- Reasoning about performance isn't trivial but definitely possible
- GHC gives us the tools to control specific aspects of our programs

Conclusion

- GHC is impressively good at optimizing high-level code
- Reasoning about performance isn't trivial but definitely possible
- GHC gives us the tools to control specific aspects of our programs
- If all else fails, GHC has a great C FFI

More Information

- The Spineless Tagless G-machine
- Detecting Space Leaks
- Inlining and Specialisation
- GHC User's Guide
- The GHC Commentary