Writing Fast Haskell
Elegance Is Not an Excuse for Bad Performance

Moritz Kiefer (@cocreature)
August 22, 2018
Haskellers often talk about elegant code.
• Haskellers often talk about elegant code
• But elegance is *not* an excuse for bad performance!
• 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
Haskell 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
• Benchmark before you optimize
• GHC supports options for time and space profiling
• Profiling can break optimizations
  • Enable profiling selectively
### Primitive Types

Correspond to “raw machine types”

E.g. **Int**, **Double**
## 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 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 \textbf{Int} type

\begin{verbatim}
data Int = I# Int#
\end{verbatim}
GHC Compilation Pipeline

Haskell

Core

STG

C--

Assembly

LLVM IR
GHC Compilation Pipeline

Haskell → Renaming, type checking, desugaring

Core

STG

C--

Assembly

LLVM IR
GHC Compilation Pipeline

Haskell → Renaming, type checking, desugaring

<table>
<thead>
<tr>
<th>Core</th>
<th>Simplifier</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>STG</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>C--</th>
<th></th>
</tr>
</thead>
</table>

- Assembly
- LLVM IR
GHC Compilation Pipeline

Haskell  →  Renaming, type checking, desugaring

   ↓

Core  →  Simplifier

   ↓

STG  →  Canonicalized core without types

   ↓

C--  →

   ↓

Assembly  →

   ↓

LLVM IR  →
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
| **let** | Allocates a thunk on the heap |
| **case** | Forces evaluation to WHNF |
Naive Sum

\[
\text{sum} :: [\text{Int}] \rightarrow \text{Int}
\]
\[
\text{sum} \; [] = 0
\]
\[
\text{sum} \; (x : xs) = x + \text{sum} \; xs
\]
Tail-Recursive Sum

\[
\text{sum} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sum} = \text{go } 0 \\
\text{where} \\
\text{go acc } [] = \text{acc} \\
\text{go acc (x : xs)} = \text{go (x + acc) xs}
\]
```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
```
{-# LANGUAGE BangPatterns #-}

sum :: [Int] -> Int

sum = go 0

  where

  go acc [] = acc

  go acc (x : xs) =
    let !acc' = x + acc
    in  go acc' xs
- `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
The following two definitions are equivalent.

\[ f \ a \ b = \ldots \]

\[ f = \ \backslash a \ b \ \ldots \]
Or are they?

\[
f(a, b) = \ldots
\]

\[
f = \backslash a, b \ldots
\]
GHC will only inline fully saturated function applications!
data Point =
  Point Int
  Int
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
```
• 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
```
Continuation Passing Style

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)
Convert

\[ f :: a \rightarrow b \]

into

\[ f :: a \rightarrow (b \rightarrow r) \rightarrow r \]

Can avoid allocations and unnecessary case distinctions
Continuation Passing Style

\[
\text{main :: IO ()} \\
\text{main} = \\
\text{loop2 100 (10, 10) $ \(x, y\) -> print (x - y)}
\]

\[
\text{loop2 :: Int -> (Int,Int) -> ((Int,Int) -> r) -> r} \\
\text{loop2 n (x, y) cont} \\
| \text{n > 0} = \text{loop2 (n - 1) (x + 1,y - 1) cont} \\
| \text{otherwise} = \text{cont (x, y)}
\]
map Fusion

map f . map g = map (f . g)
### map Fusion

\[ \text{map } f \circ \text{map } g = \text{map} \ (f \circ g) \]

### build/foldr Fusion

\[
\begin{align*}
\text{build} &:: (\forall b. (a \to b \to b) \to b \to b) \\
&\to [a] \\
\text{foldr} &:: (a \to b \to b) \to b \to [a] \to b \\
\text{foldr } f a (\text{build } g) &= g \ f \ a
\end{align*}
\]
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
<table>
<thead>
<tr>
<th>Array#</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Array of boxed values</td>
</tr>
<tr>
<td></td>
<td>• Card table to avoid having to scan unmodified entries in GC</td>
</tr>
<tr>
<td>SmallArray#</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Array of boxed values</td>
</tr>
<tr>
<td></td>
<td>• No card table</td>
</tr>
<tr>
<td>ByteArray#</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Region of raw memory</td>
</tr>
<tr>
<td></td>
<td>• Pinned and unpinned</td>
</tr>
</tbody>
</table>
• **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 hashables from the **hashables** package are often slower
• 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