Haskell Performance Patterns

Johan Tibell

February 15, 2012

Caveats

- Much of this is GHC specific.
- Some of the patterns trade generality/beauty for performance. Only use these when needed.
- The following patterns are guidelines, not rules. There are exceptions.

Think about your data representation

- A linked-list of linked-lists of pointers to integers is not a good way to represent a bitmap! (Someone actually did this and complained Haskell was slow.)
- Make use of modern data types: `ByteString`, `Text`, `Vector`, `HashMap`, etc.

Unpack scalar fields

Always unpack scalar fields (e.g. `Int`, `Double`):

```
data Vec3 = Vec3 {-# UNPACK #-} !Double
                 {-# UNPACK #-} !Double
                 {-# UNPACK #-} !Double
```

- This is **the most important optimization available** to us.
- GHC does Good Things (tm) to strict, unpacked fields.
- You can use `-funbox-strict-fields` on a per file basis if UNPACK is too verbose.

Use a strict spine for data structures

- Most container types have a strict spine e.g. `Data.Map`:

```
data Map k a = Tip
             | Bin {-# UNPACK #-} !Size !k a
                   !(Map k a) !(Map k a)
```

(Note the bang on the `Map k a` fields.)

- Strict spines cause more work to be done up-front (e.g. on insert), when the data structure is in cache, rather than later (e.g. on the next lookup.)

- Does not always apply (e.g. when representing streams and other infitinte structures.)

Specialized data types are sometimes faster

- Polymorphic fields are always stored as pointer-to-thing, which increases memory usage and decreases cache locality. Compare:

```
data Tree a = Leaf | Bin a !(Tree a) !(Tree a)
data IntTree = IntLeaf | IntBin {-# UNPACK #-} !Int !IntTree !IntTree
```

- Specialized data types can be faster, but at the cost of code duplication. **Benchmark** your code and only use them if really needed.

Inline recursive functions using a wrapper

- GHC does not inline recursive functions:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

- **If** you want to inline a recursive function, use a non-recursive wrapper like so:

```
map :: (a -> b) -> [a] -> [b]
map f = go
  where
    go []     = []
    go (x:xs) = f x : go xs
```

- You still need to figure out if you want a particular function inlined (e.g. see the next slide.)

Inline HOFs to avoid indirect calls

- Calling an unknown function (e.g. a function that's passed as an argument) is more expensive than calling a known function. Such *indirect* calls appear in higher-order functions:

```haskell
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

g xs = map (+1) xs  -- map is recursive => not inlined
```

- At the cost of increased code size, we can inline `map` into `g` by using the non-recursive wrapper trick on the previous slide together with an `INLINE` pragma.

- Inline HOFs if the higher-order argument is used a lot (e.g. in `map`, but not in `Data.Map.insertWith`.)

Use strict data types in accumulators

If you're using a composite accumulator (e.g. a pair), make sure it has strict fields.

Allocates on each iteration:

```haskell
mean :: [Double] -> Double
mean xs = s / n
  where (s, n) = foldl' (\ (s, n) x -> (s+x, n+1)) (0, 0) xs
```

Doesn't allocate on each iteration:

```haskell
data StrictPair a b = SP !a !b

mean2 :: [Double] -> Double
mean2 xs = s / n
  where SP s n = foldl' (\ (SP s n) x -> SP (s+x) (n+1)) (SP 0 0) xs
```

Haskell makes it cheap to create throwaway data types like `StrictPair`: one line of code.

Use strict returns in monadic code

`return` often wraps the value in some kind of (lazy) box. This is often not what we want, especially when returning some arithmetic expression. For example, assuming we're in a state monad:

```
return $ x + y
```

creates a thunk. We most likely want:

```
return $! x + y
```

Beware of the lazy base case

Functions that would otherwise be strict might be made lazy by the "base case":

```
data Tree = Leaf
          | Bin Key Value !Tree !Tree

insert :: Key -> Value -> Tree
insert k v Leaf = Bin k v Leaf Leaf  -- lazy in @k@
insert k v (Bin k' v' l r)
   | k < k'    = ...
   | otherwise = ...
```

Since GHC does good things to strict arguments, we should make the base case strict, unless the extra laziness is useful:

```
insert !k v Leaf = Bin k v Leaf Leaf  -- strict in @k@
```

In this case GHC might unbox the key, making all those comparisons cheaper.

Beware of returning expressions inside lazy data types

- Remember that many standard data types are lazy (e.g. `Maybe`, `Either`).
- This means that it's easy to be lazier than you intend by wrapping an expression in such a value:

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just $ x / y  -- creates thunk
```

- Force the value (e.g. using `$!`) before wrapping it in the constructor.

Summary

- Strict fields are good for performance.
- Think about your data representation (and use UNPACK where appropriate.)