SpecConstr: optimising purely functional loops

Amos Robinson

September 16, 2013

Compiler divergence

Compile this program with `-O2`, and the compiler hangs!

```
import Data.Vector as V
reverseV = V.foldl (flip (:)) []
```

(Use GHC < 7.7)

Dot product

The code we want to write

```
dotp :: Vector Int -> Vector Int -> Int
dotp as bs
  = fold    (+) 0
  $ zipWith (*) as bs
```

Dot product

The code we want to run

```
dotp as bs = go 0 0
 where
  go i acc
   | i > V.length as
   = acc
   | otherwise
   = go (i + 1) (acc + (as!i * bs!i))
```

No intermediate vectors, no constructors, no allocations: perfect.
(Just pretend they're not boxed ints...)

Dot product

The code we get after stream fusion (trust me)

```
dotp as bs = go (Nothing, 0) 0
 where
  go (_, i) acc
   | i > V.length as
   = acc
  go (Nothing, i) acc
   = go (Just (as!i), i) acc
  go (Just a, i) acc
   = go (Nothing, i + 1) (acc + (a * bs!i))
```

All those allocations!

Dot product

The code we get after stream fusion (trust me)

```
dotp as bs = go (Nothing, 0) 0
 where
  go (_, i) acc
   | i > V.length as
   = acc
  go (Nothing, i) acc
   = go (Just (as!i), i) acc
  go (Just a, i) acc
   = go (Nothing, i + 1) (acc + (a * bs!i))
```

Only to be unboxed and scrutinised immediately. What a waste.

Dot product

Let us try specialising this by hand.

```
dotp as bs = go (Nothing, 0) 0
 where







  go (_, i) acc
   | i > V.length as  = acc
  go (Nothing, i) acc = go (Just (as!i), i) acc
  go (Just a,  i) acc = go (Nothing, i+1) (acc + (a*bs!i))
```

Start by looking at the first recursive call. We can specialise the function for that particular call pattern.

Dot product

Let us try specialising this by hand.

```
dotp as bs = go'1 0 0
 where
  go'1 i acc = case i > V.length as of
   True  -> acc
   False -> go (Just (as!i), i) acc



  go (_, i) acc
   | i > V.length as  = acc
  go (Nothing, i) acc = go (Just (as!i), i) acc
  go (Just a,  i) acc = go'1 (i + 1) (acc + (a * bs!i))
```

Specialise on `go (Nothing, x) y = go'1 x y`

Dot product

Let us try specialising this by hand.

```
dotp as bs = go'1 0 0
 where
  go'1 i acc = case i > V.length as of
   True  -> acc
   False -> go (Just (as!i), i) acc




  go (_, i) acc
   | i > V.length as  = acc
  go (Nothing, i) acc = go (Just (as!i), i) acc
  go (Just a,  i) acc = go'1 (i + 1) (acc + (a * bs!i))
```

Now look at the call in the new function. We can specialise on that pattern, too!

Dot product

Let us try specialising this by hand.

```
dotp as bs = go'1 0 0
 where
  go'1 i acc = case i > V.length as of
   True  -> acc
   False -> go'2 (as!i) i acc
  go'2 a i acc = case i > V.length as of
   True  -> acc
   False -> go'1 (i + 1) (acc + (a * bs!i))
  go (_, i) acc
   | i > V.length as  = acc
  go (Nothing, i) acc = go (Just (as!i), i) acc
  go (Just a,  i) acc = go'1 (i + 1) (acc + (a * bs!i))
```

Specialise on `go (Just x, y) z = go'2 x y z`

Dot product

Let us try specialising this by hand.

```
dotp as bs = go'1 0 0
 where
  go'1 i acc = case i > V.length as of
   True  -> acc
   False -> go'2 (as!i) i acc
  go'2 a i acc = case i > V.length as of
   True  -> acc
   False -> go'1 (i + 1) (acc + (a * bs!i))
  go (_, i) acc
   | i > V.length as  = acc
  go (Nothing, i) acc = go (Just (as!i), i) acc
  go (Just a,  i) acc = go'1 (i + 1) (acc + (a * bs!i))
```

Now it turns out that go isn't even mentioned any more. Get rid of it.

Dot product

Let us try specialising this by hand.

```
dotp as bs = go'1 0 0
 where
  go'1 i acc = case i > V.length as of
   True  -> acc
   False -> go'2 (as!i) i acc
  go'2 a i acc = case i > V.length as of
   True  -> acc
   False -> go'1 (i + 1) (acc + (a * bs!i))
```

These two are mutually recursive, but we can still inline `go'2` into `go'1`.

Dot product

Let us try specialising this by hand.

```
dotp as bs = go'1 0 0
 where
  go'1 i acc = case i > V.length as of
   True  -> acc
   False -> case i > V.length as of
      True  -> acc
      False -> go'1 (i + 1) (acc + (as!i * bs!i))
```

The case of `i > V.length as` is already inside the False branch of a case of the same expression, we can remove the case altogether.

Dot product

Let us try specialising this by hand.

```
dotp as bs = go'1 0 0
 where
  go'1 i acc = case i > V.length as of
   True  -> acc
   False -> go'1 (i + 1) (acc + (as!i * bs!i))
```

Which was what we wanted.

ForceSpecConstr

SpecConstr puts a limit on the number of specialisations, as too many specialisations causes code blowup.

```
unstream :: Stream a -> [a]
unstream (Stream f s) = go ForceSpecConstr s
 where
  go ForceSpecConstr s
   = case f s of
    Done        -> []
    Skip    s' ->       go ForceSpecConstr s'
    Yield a s' -> a :   go ForceSpecConstr s'
```

But with stream fusion, such as in the `vector` library, we want to specialise everything no matter what.

ForceSpecConstr termination

A nasty bug in ForceSpecConstr meant that specialising on recursive types would produce infinite specialisations.

```
reverse :: [a] -> [a]
reverse as = go ForceSpecConstr as []
 where
  go []     acc = acc
  go (a:as) acc = go as (a:acc)

SPECIALISE go as (a:acc):
go'1 as a acc
 = case as of
     [] -> (a:acc)
     (a':as') -> go as' (a':a:acc)

SPECIALISE go as' (a':a:acc):
go'2 as' a' a acc
 = ...
```

ForceSpecConstr termination

This is fixed simply by limiting specialisation on recursive types a fixed number of times (`-fspec-constr-recursive`).

Seeding of specialisation

Looking at the function body, there are many call patterns to specialise.

```
go :: Either (Maybe Int) (Maybe Int) -> Int
go c = case c of
 Left    a ->
    go $ Right a
 Right  Nothing ->
    go $ Left Nothing
 Right (Just a) ->
    go $ Left (Just (a-1))
```

There are three patterns here to specialise on. After specialising, we get two more patterns.

Seeding of specialisation

If it is a local, non-exported function, we are much better off starting specialisation with calls from the current module.

```
go :: Either (Maybe Int) (Maybe Int) -> Int
go c = case c of
 Left    a ->
    go $ Right a
 Right  Nothing ->
    go $ Left Nothing
 Right (Just a) ->
    go $ Left (Just (a-1))

main =
 putStrLn $ go $ Right Nothing
```

Here, we produce only two specialisations: initially `Right Nothing`, and then `Left Nothing`.

End

end.