SpecConstr: optimising purely functional loops

Amos Robinson
PhD student at UNSW

September 22, 2013

Compiler divergence

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

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

```
> ghc -O2 -v TestReverse.hs
...
*** Simplifier:
Result size of Simplifier
  = terms: 60, types: 60, coercions: 16
*** SpecConstr:
Result size of SpecConstr
```

(non termination)

Code blowup in stream fusion

```
let xs = enumFromTo 1 len
in        (xs ++ xs) `zip` (xs ++ xs)
    `zip` (xs ++ xs) `zip` (xs ++ xs)
    `zip` (xs ++ xs) `zip` (xs ++ xs)
```

```
> ghc -O2 -v Blowup.hs
...
*** Simplifier:
Result size of Simplifier
  = terms: 678, types: 2,594, coercions: 9
*** SpecConstr:
Result size of SpecConstr
  = terms: 119,108, types: 415,625, coercions: 9
```

(21 seconds)

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!

Constructor specialisation

1. Find all recursive calls in go

```
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))
```

So-called interesting call patterns.

Constructor specialisation

2. Create a copy of go for each call pattern

```
go (Nothing, y) z = go'1   y z
go'1 i acc
 | i > V.length as = acc
 | otherwise       = go (Just (as!i), i) acc

go (Just  x, y) z = go'2 x y z
go'2 a i acc
 | i > V.length as = acc
 | otherwise       = go (Nothing, i + 1) (acc + (a * bs!i))
```

Then find any new call patterns in the new functions' bodies.

Constructor specialisation

3. Apply rewrite rules for each pattern

```
go (Nothing, y) z = go'1   y z
go'1 i acc
 | i > V.length as = acc
 | otherwise       = go'2     (as!i)  i  acc

go (Just  x, y) z = go'2 x y z
go'2 a i acc
 | i > V.length as = acc
 | otherwise       = go'1          (i + 1) (acc + (a * bs!i))
```

After SpecConstr

Normal optimisation resumes. go is dead.

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

  go'1 i acc
   | i > V.length as  = acc
   | otherwise        = go'2 (as!i) i acc

  go'2 a i acc
   | i > V.length as  = acc
   | otherwise        = go'1 (i + 1) (acc + (a * bs!i))
```

After SpecConstr

We can inline go'2 into go'1 and remove the superfluous case.

```
dotp as bs = go'1 0 0
 where






  go'1 i acc
   | i > V.length as   = acc
   | otherwise         = go'2 (as!i) i acc

  go'2 a i acc
   | i > V.length as   = acc
   | otherwise         = go'1 (i + 1) (acc + (a * bs!i))
```

After SpecConstr

And we have the ideal result.

```
dotp as bs = go'1 0 0
 where









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

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

I fixed this simply by limiting specialisation on recursive types a fixed number of times.
There's a compiler option for this:

```
>ghc -fspec-constr-recursive=3
```

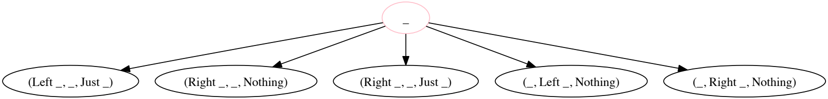
Complicated program

```
-- go = ([0..2] ++ [0..3]) `zip` ([0..3] ++ [0..2])

go :: (Either Int Int, Either Int Int, Maybe Int) -> [Int]
go (Left i, z, Nothing)
 | i <= 2
 = go (Left (i+1), z, Just i)
 | otherwise
 = go (Right 0,    z, Nothing)
go (Right i, z, Nothing)
 | i <= 3
 = go (Right (i+1), z, Just i)
 | otherwise
 = []
go (y, Left  j, Just i)
 | j <= 3
 = (i, z)
 : go (y, Left (j+1), Nothing)
 | otherwise
 = go (y, Right 0,    Nothing)
go (y, Right j, Just i)
 | j <= 2
 = (i, z)
 : go (y, Right (j+1), Nothing)
 | otherwise
 = []

main = putStrLn $ show $ go (Left 0, Left 0, Nothing)
```

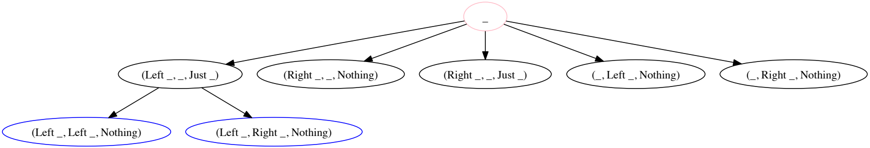
Specialisation graph - 1



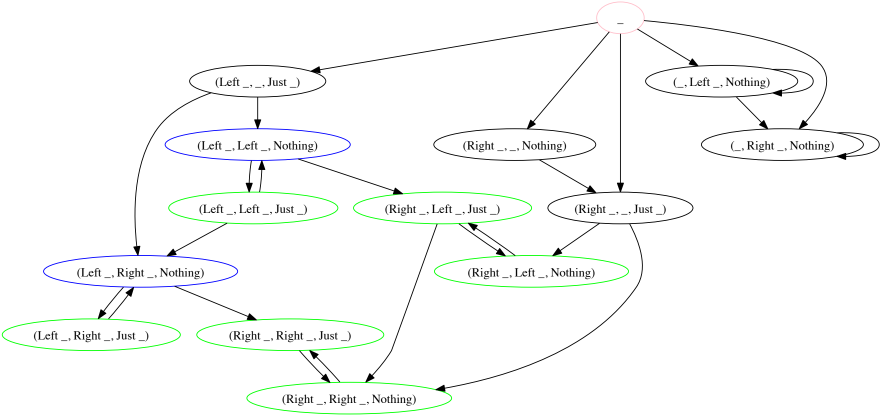
Specialisation on (Left _, _, Just _)

```
go :: (Either Int Int, Either Int Int, Maybe Int) -> [Int]
go (Left i, z, Nothing)
 | i <= 2
 = go (Left (i+1), z, Just i)
 | otherwise
 = go (Right 0,     z, Nothing)
go (Right i, z, Nothing)
 | i <= 3
 = go (Right (i+1), z, Just i)
 | otherwise
 = []
go (Left y, Left  j, Just i)
 | j <= 3
 = (i, z)
 : go (Left y, Left (j+1), Nothing)
 | otherwise
 = go (Left y, Right 0,     Nothing)
go (Left y, Right j, Just i)
 | j <= 2
 = (i, z)
 : go (Left y, Right (j+1), Nothing)
 | otherwise
 = []
```

Specialisation graph - 2



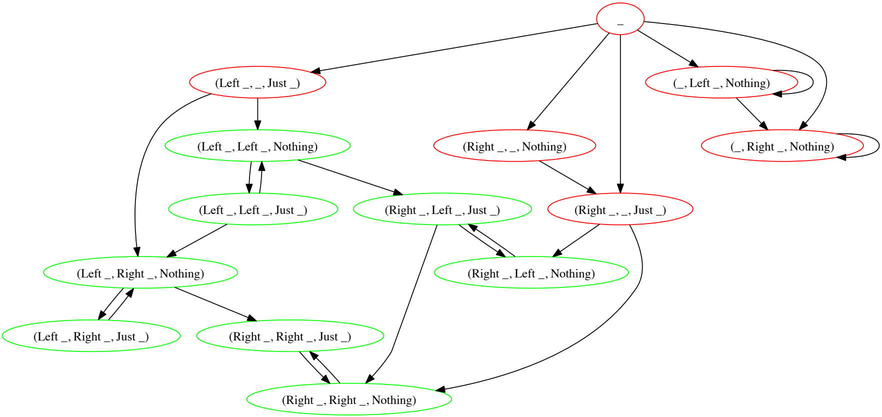
Specialisation graph - 3



Seeding

```
main = putStrLn $ show $ go (Left 0, Left 0, Nothing)
```

Seeding of specialisation



Seeding

- Already done for local `let`-defined functions
- But local `let` functions can be lifted by simplifier!

Seeding requirements

- Only works if go is not exported

Otherwise, calls from other modules could use other call patterns.

Seeding requirements

- All call patterns must be *interesting*

Same as for `let`-defined functions.

Code blowup - benchmark

```
let xs = enumFromTo 1 len
in       (xs ++ xs) `zip` (xs ++ xs)
   `zip` (xs ++ xs) `zip` (xs ++ xs)
   `zip` (xs ++ xs) `zip` (xs ++ xs)
```

Before

```
Result size of SpecConstr
  = terms: 119,108, types: 415,625, coercions: 9
(21 seconds)
```

After

```
Result size of SpecConstr
  = terms: 29,372, types: 94,772, coercions: 9
(3 seconds)
```

End

end.