SpecConstr: optimising purely functional loops

Amos Robinson

May 30, 2013

Motivation - dot product

The code we want to write

```
type V = Unboxed.Vector

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

Motivation - 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...)

Motivation - 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!

Motivation - 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.

Motivation - 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.

Motivation - 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`

Motivation - 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!

Motivation - 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`

Motivation - 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.

Motivation - 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.

Motivation - 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.

Motivation - 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.

GHC pipeline (not to scale)

We now have some intuition about SpecConstr. How does it fit in with the rest of GHC's optimisations?

Parse	::	String	$\rightarrow$	Source
Typecheck	::	Source	$\rightarrow$	Source
Desugar	::	Source	$\rightarrow$	Core
Simplify	::	Core	$\rightarrow$	Core
SpecConstr	::	Core	$\rightarrow$	Core
Simplify $\times$ 50	::	Core	$\rightarrow$	Core
Code generation	::	Core	$\rightarrow$	Object

GHC pipeline (not to scale)

We now have some intuition about SpecConstr. How does it fit in with the rest of GHC's optimisations?

Parse	::	String	$\rightarrow$	Source
Typecheck	::	Source	$\rightarrow$	Source
Desugar	::	Source	$\rightarrow$	Core
Simplify	::	Core	$\rightarrow$	Core
SpecConstr	::	Core	$\rightarrow$	Core
Simplify $\times$ 50	::	Core	$\rightarrow$	Core
Code generation	::	Core	$\rightarrow$	Object

Focus on these parts.

Simplifier

The simplifier does a bunch of transforms in a single pass:

- Case of constructor
- Inlining
- Rewrite rules
- Let floating
- Beta reduction

and many more, but these are the most interesting for us

Simplifier

Case of constructor

```
case (Just a) of
 Nothing -> x
 Just a' -> y

==>

let a' = a
in  y
```

When the scrutinee of a case is known to be a constructor, we can remove the case altogether.

Simplifier

Inlining

```
zipWith f xs ys
 = unstream $ zipWith_S f
   (stream xs) (stream ys)
...
zipWith (*) as bs


==>

...
unstream $ zipWith_S (*)
(stream as) (stream bs)
```

Move the definition of a function into places it is used

Simplifier

Rewrite rules

```
{-# RULES stream (unstream xs) = xs #-}

fold_S (+) $ stream $ unstream $
zipWith_S (*) (stream as) (stream bs)
==>
...
fold_S (+) $
zipWith_S (*) (stream as) (stream bs)
```

Replace left-hand side with right, anywhere

SpecConstr, actually

The basic idea:

- Find calls with constructors
- Create new functions for that call pattern
- Add rewrite rules for each call pattern
- Let the simplifier do the rest

```
enumFromTo f t acc
 = case f > t of
   True  -> acc
   False -> enumFromTo f (t-1) (t : acc)
```

(Silly example.)

SpecConstr, actually

The basic idea:

- Find calls with constructors
- Create new functions for that call pattern
- Add rewrite rules for each call pattern
- Let the simplifier do the rest

```
enumFromTo f t acc
 = case f > t of
   True  -> acc
   False -> enumFromTo'1 f (t-1) t acc
enumFromTo'1 f t cons acc
 = case f > t of
   True  -> acc
   False -> enumFromTo f (t-1) (t : cons : acc)
```

Not only will this diverge, it's not even decreasing allocations!

SpecConstr, actually

The basic idea:

- Find calls with constructors on scrutinised parameters
- Create new functions for that call pattern
- Add rewrite rules for each call pattern
- Let the simplifier do the rest

```
enumFromTo f t acc
 = case f > t of
   True  -> acc
   False -> enumFromTo f (t-1) (t : acc)
```

SpecConstr, actually

Looking through bindings

```
silly2 xs' = case xs' of
 []      -> []
 (x:xs) -> if   x > 10
           then (do1 (x:xs), do2 (x:xs)) : silly2 (x:xs)
           else silly2 xs
```

Common subexpression elimination (CSE) will probably rewrite those `x:xs` into `xs'`.

SpecConstr, actually

Looking through bindings

```
silly2 xs' = case xs' of
 []      -> []
 (x:xs) -> if   x > 10
           then (do1    xs', do2    xs') : silly2    xs'
           else silly2 xs
```

But now it's not obvious that `silly2 xs'` is a valid call pattern. No matter: keep track of the bound variables and their values. If we know `xs'=x:xs`, we can still specialise.

SpecConstr, actually

Reboxing

```
silly2 xs' = case xs' of
 []      -> []
 (x:xs) -> if   x > 10
           then (do1    xs', do2    xs') : silly2    xs'
           else silly2 xs
```

Now we'll specialise on `silly2 (x:xs) = silly2'1 x xs`.

SpecConstr, actually

Reboxing

```
silly2 xs' = case xs' of
 []      -> []
 (x:xs) -> if   x > 10
           then (do1    xs', do2    xs') : silly2    xs'
           else silly2 xs
silly2'1 x xs
 =         if   x > 10
           then (do1 (x:xs), do2 (x:xs)) : silly2'1 x xs
           else silly2 xs
```

Hey! Now we're actually doing *more* allocations.
The moral: don't specialise on a bound variable if the variable is used elsewhere.

ForceSpecConstr

SpecConstr puts a limit on the number of specialisations, to prevent 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, we want to specialise everything no matter what. Damn the consequences!

End

end.