Type Inference and Optimisation for an Impure World

Ben Lippmeier

Australian National University

Horray for Compositional Programming

```
paths = map    (\name -> "/" ++ name)
      $ filter (\name -> head name /= '.')
      $ map toLower files
```

Lists, lambda abstractions and combining forms.

All is right with the world.

It's all fun and games until IO gets involved...

This doesn't work, the types are all wrong.

```
map putStr $ map (++ "\n")
           $ filter (\name -> head name /= '.')
           $ getDirectoryContents "."
```

```
filter :: (a -> Bool) -> [a] -> [a]
getDirectoryContents
       :: String -> IO [String]
```

Forever lifting...

```
( liftM (map (++ "\n"))
$ liftM (filter (\name -> head name /= '.'))
$ getDirectoryContents ".")
>>= \ff -> mapM_ putStr ff
```

- Argh @ IO
- The wrappers needed to use state monads are cluttering up my real programs.

A lack of data dependencies.

```
readSub :: () -> Int
readSub ()
 = let a = readInt ()
       b = readInt ()
   in  a - b
```

... but what is forcing the two let bindings to be evaluated in the right order?
(Answer: not much).

- Problems for complier optimisations.
- Problems for lazy evaluation.

Code motion.

```
map f (map g xs) = map (f . g) xs
```

- Converting the first form to the second eliminates the intermediate list.
- It's a specific example of a 'code motion' style optimisation.
- Others are let-floating, full-laziness, case merging, deforestation...
- *many* such optimisations are used in GHC.

... but it only works for pure expressions.

```
let a = 0
in  map (\x -> printInt a)
  $ map (\x -> a := a + x) [1..100]
```

Does *not* do the same thing as:

```
let a = 0
in  map ((\x -> printInt a) . (\x -> a := a + x))
        [1..100]


print :: Int -> ()
(:=)  :: Int -> Int -> ()
```

Solution 1: Thread the world

We *could* introduce the required data dependency by threading a dummy *state token* through our program.

```
readSubW :: () -> World -> (World, Int)
readSubW () w0
 = let (w1, a) = readIntW () w0
       (w2, b) = readIntW () w1
   in  (w2, a - b)
```

This works, but it changes the shape of our function's type and clutters the program. (This is what Clean does)

Solution 2: Hide the state token in a monad

We *could* hide the state token behind a type definition and some combinators, but it doesn't really solve the problem.

```
data World
type IO a = World -> (World, a)

readSubM :: () -> IO Int
readSubM ()
 = do   a  <- readIntM ()
        b  <- readIntM ()
        return (a - b)
```

The price of state monads.

- `readSubM` now has a different structural type compared with our original `readSub` function.
- Monadic functions don't compose well with non-monadic functions.
- do-notation looks a lot like a `(let ... in ...)` expression, but not the same.

Haskell has stratified into ‘pure’ and monadic sub-languages...

```
map       :: (a -> b)     -> [a] ->    [b]
mapM      :: Monad m
          => (a -> m b) -> [a] -> m [b]

foldl     :: (a -> b ->   a) ->  a  -> [b] ->    a
foldM     :: Monad m
          => (a -> b -> m a) ->  a  -> [b] -> m  a

zipWith   :: (a -> b ->   c) -> [a] -> [b] ->   [c]
zipWithM  :: Monad m
          => (a -> b -> m c) -> [a] -> [b] -> m [c]
```

... with missing versions of monadic functions.

```
find ::  (a -> Bool) -> [a] -> Maybe a
```
...but no `findM`!

```
any  ::  (a -> Bool) -> [a] -> Bool
```
...but no `anyM`!

```
span ::  (a -> Bool) -> [a] -> ([a], [a])
```
...but no `spanM`!

- All higher order functions need a monadic version.
  (What a hassle!)
- If you start to write ‘pure’ code and realise you needed a monad half way through then you’re in for a large amount of refactoring.
- You might as well write all code monadically right from the start!

What do we really want?

- Code that uses computational effects should compose well with pure code.
- We would like the effects to show up in the types of our functions.
- Adding a piece of effectful code to existing pure code shouldn't require a huge amount of refactoring.
- We need some way of maintaining evaluation order between effectful function applications.
- We would like to support laziness if possible.

Effect inference to the rescue.

```
readSub :: () -(!e1)> Int
        :- !e1 = {!Console};
readSub ()
 = do   a = readInt ()
        b = readInt ()
        a - b
```

- Effect information is orthogonal to the ‘shape’ information in the types.
- We don’t need separate `let` and `do` binding forms.
- We support destructive update of arbitrary data structures. The type system tracks the resulting effects, and the optimisations cope.

Types, Regions, Effects and Closures.

This looks complicated, but as the region effect and closure information is orthogonal to the shape, it can be (mostly) elided by the programmer.

```
map :: forall t0 t1 %r0 %r1 !e0 !e1 $c0 $c1
    .  (t0 -(!e1 $c1)> t1)
           -> List %r1 t0 -(!e0 $c0)> List %r0 t1
    :- !e0 = !{ !Read %r1; !e1 }
    ,  $c0 = ${ $c1 }

map f []       = []
map f (x:xs)   = f x : map f xs
```

A Lazy map.

```
mapL :: forall t0 t1 %r0 %r1 !e0 !c0 !c1
     .  (t0 -(!e0 $c1)> t1)
          -> List %r1 t0 -($c0)> List %r0 t1
     :- $c0 = ${ $c1 }
     ,  Lazy %r0
     ,  Pure !e0
     ,  Const %r1

mapL f []       = []
mapL f (x:xs)   = f x : mapL @ f x
```

Demos...

`List.ds`

`List.dump-type-constraints.dc`

`List.dump-core-lifted.dc` - System-F, map and mapL

`List.dump-core-curry.dc` - Apply, Curry, Call

`List.dump-sea-source.dc`

`List.ddc.c`

`Graphics/Simple`

`Graphics/N-Body.ds` - destructive update.

`Vec2.ds` - field projections.