Type inference and optimisation for an impure world.

Ben Lippmeier

Australian National University

From **Wikipedia**:

Functional programming is a programming paradigim that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

an empty definition?

Mutable state is useful

- Some ”pure functional programs” are based *totally* around state and mutable data.
- Some programs *need* mutable data for efficiency.
- Mutable state is a convenient feature in the programming model.

From GHC:

```
data TcTyVarDetails
   = SkolemTv SkolemInfo
   | MetaTv BoxInfo (IORef MetaDetails)
                     ^^^^^
```

Unsupported features:

The ability to write programs with computational effects and mutable state is a *feature* of a language.

- **But** -

These features create headaches for compiler writers and people into formal semantics.

- Changing the order of interfering effects can change the meaning of a program.
- Changing the sharing properties of mutable data can change the meaning of a program.

Solutions?

- Separate programs into “pure” and “impure”.
- Disparage the impure ones.
- Wrap a state monad around impure code and call it pure.
- Feel satisfied.

Works well on `haskell-cafe`!

Yay for state monads

- State monads help us thread world tokens through our program so we can express the data dependencies which are not otherwise visible to the compiler.
- You can erase the world tokens before native code generation so the program doesn't suffer a performance loss.
- The effect that a piece of code has is expressed in its type.

Boo at state monads

- Monadic code does not compose well with non-monadic code.
- You need pure and monadic versions of every higher-order function.
- Haskell has stratified into “pure” and monadic sublanguages.

```
fun ()                   fun' ()
 = let x = f ...          = do let x = f ...
       y = map g x               y    <- mapM g' x
   in  y                         return y

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

Another solution?

- Allow the programmer to use arbitrary computational effects.
- Have the compiler infer which data is mutable and which function applications cause effects.
- Annotate the intermediate language with this information and use this to guide the optimisations.

Example: `map.core.ds`

- Compiler can now reason about effects directly.
- Effect information in types is orthogonal to shape/structure information.

Types, Regions, Effects, Closures ...

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

map f Nil              = Nil
map f (Cons x xs)      = Cons (f x) (map f xs)
```

... and Classes

```
updateInt
        :: Mutable %r1
        => Int %r1 -> Int %r2 -(!e1 $c1)> ()
        :- !e1 = !{ !Read %r2; !Write %r1; }
        ,  $c1 = ${ Int %r1 }

suspend
        :: Pure !e1
        => (a -(!e1)> b) -> a -> b;
```

Play together nicely now, kids.

```
fun2 ()
 = do { list1    = [1..];
        list2    = mapL ((*) 2) list1;
        ...
        (head list1) := 5;
        ...
 };
```

```
mapL :: (Pure !e1, Const %r0)
     => (a -(!e1)> b) -> List %r0 a -> List %r1 b
```

```
mapL f []       = []
mapL f (x:xs)   = suspend1 f x : suspend2 mapL f xs
```

```
test/Error/CheckConst/PureReadWrite/Main.ds:15:21
    Cannot write to Const region.
      This region is being forced Const because there is a
      purity constraint on a Read effect which accesses it.
                effect: !Write @165
             caused by: (:=)
                    at: Main.ds:15:21

         conflicts with,
                effect: !Read @165
             caused by: (*)
                    at: Main.ds:14:25

        which is being purified by,
            constraint: Base.Pure @230
       from the use of: mapL
                    at: Main.ds:14:18
```

Of course, there are issues with type inference..

```
printInt
        :: forall %r1
        .  Int %r1 -(!e1)> ()
        :- !e1 = !{ !Read %r1; !Console; };

fun f  = if ... then f else printInt

fun   :: forall %r1
        .  (Int %r1 -(!e1)> ()) -> Int %r1 -(!e1)> ()
        :- !e1 = !{ !Read %r1; !Console; };
```

Uh oh. What does the first `!e1` in the type for `fun` mean?

Rewrites

Region/effect/closure variables in a contra-variant branch are *always* inputs - ie they do not represent constraints on what that particular variable can be. We can rewrite to the desired form.

```
fun   ::  forall %r1
      .   (Int %r1 -(!e1)> ()) -> Int %r1 -(!e1)> ()
      :-  !e1 = !{ !Read %r1; !Console; };
```

rewrites to:

```
fun   ::  forall %r1 %r2 !e1
      :-  (Int %r1 -(!e1)> ()) -> Int %r2 -(!e2)> ()
      ,   !e2 = !{ !Read %r3; !Console; !e1}
      ,   %r3 = %{ %r1; %r2 }
```

Bi-directional unification is not the right operation

```
(==) :: forall a %r1
      .  Eq a
     => a -> a -(!e1 $c1)> Bool %r1
     :- !e1 = !Read a,  $c1 = (x : a);

x1 :: Const   %r5 => Int %r5;
x2 :: Mutable %r6 => Int %r6;

y  = (x1 == x2)
```

`%r5` and `%r6` are being forced to be the same via the type variable `a` – but a region can't be both Const and Mutable at the same time.

Shape Constraints

```
(==) :: forall a b %r1
     .  (Eq a, Shape a b)
     => a -> b -(!e1 $c1)> Bool %r1
     :- !e1 = !{ !Read a; !Read b; }
     ,  $c1 = (x : a)
```

`Shape` forces `a` and `b` to have the same *structure*, without placing any constraint on regions, effects or closures.

This is in the same spirit as the type equality witnesses in F_c – the constraint is maintained during type inference and in the Core IR but no dictionary is passed at runtime.

Demos

- `n-body`
- `spinner`
- `Bresenham`