Graph based Type, Region, Effect, Closure, Mutability, Purity and Escape inference

- or -

Drowning in Kinds

Ben Lippmeier

Australian National University

Recap from last time

- Many useful programs make use of *state*.
- In a language with effects, we can use destructive update to modify the state.
- In a language without effects, we could:
  - manually thread the state through the program.
  - use a state monad to hide the state threading.
    (`Control.Monad.State`)
  - use a state monad to introduce the required data dependencies, then do the destructive update anyway.
    (`IO` / `IORef` / `IOArray`)

Problems:

- Untracked effects are bad news for many useful optimisations. (eg full laziness, deforestation, let floating).
- Manual threading of state is tedious and error prone.
- Monadic code does not compose well with non-monadic code.

```
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
```

Effect inference

Allow the programmer to use arbitrary computational effects, then have the compiler infer where they are so it can optimise around them.

Example: `map.dump-core.ds`

Regions

- A function which uses destructive update on local data should be compatable with one that doesn't, so long as the effects are not visible to the caller.
- We need to differentiate local *vs* non-local data.
- Add a region variable to every data constructor. Use universal quantification of regions to denote freshness of data.
- We don't use regions on function constructors, because they can't be updated.
- We can perform effect masking at generalisation time.

```
fun1 x
 = do   y     = x;
        y2    = y  + x;
        y3    = y2 + 1;
        y3;
```

```
fun2 x
 = do   y    = x;
        y    := y + x;
        y3   =  y + 1;
        y3;
```

- `fun1` has no effects.

```
fun1 :: forall %r1 %r2
      .  Int %r1 -> Int %r2
```

- `fun2` has effects, but they are not visible in the function's shape.

```
fun2 :: Int %r1 -(!e1)> Int %r3
      :- !e1 = !{ !Read %r2; !Write %r2 };
```

Effects on `%r2` can be masked, then the type generalised to be the same as `fun1`.

Closures

A function might return a reference to data in its closure. This data is shared between all applications of the function, but is not visible in its shape or type environment.

```
fun1 :: forall %r1. () -> () -> Int %r1
fun1 ()
 = let x = 5
   in  \() -> x

fun2 :: forall %r1. () -> Int %r1
fun2 = fun1 ()
```

The type for `fun2` is here is wrong, `%r1` is not fresh, it's part of the closure of the inner function.

A more descriptive type for `fun1` would be:

```
fun1 :: forall %r1
     .  () -> () -($c1)> Int %r1
     :- $c1 = ${ x : Int %r1 }
```

Now, when we come to generalise the type for `fun2` we can avoid quantifying regions which are free in the closure of the outermost function.

```
fun2 :: () -($c1)> Int %r1
     :- $c1 = ${ x : Int %r1 }
```

This problem is related to the one concerning the unsoundness of Hindley-Milner style polymorphic type inference, ie the one that the value restriction (and Leroy's closure typing) solves.

Mutablity, Constness and Purity

- Reading an object which may change in the future is a computational effect, and must be tracked.
- If you suspend a function application which is going to read data, then that data should be immutable, else the result will be dependant on when it is forced. (not *usually* what you want)
- We want to be able to write code which operates on both mutable and immutable data structures $\Rightarrow$ no `Ref` or `IORef` constructors!.

We can represent Mutablity, Constness and Purity as type-class-esque constraints on regions and effects.

```
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;
```

Example

Show `map.ds`

P is for pathological

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

fun f  = if b 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?

The problem is that `!e1` isn't quantified, but we want to allow arbitrary functions to be passed in for the first parameter.

If we treat `!e1` as an extension variable and `:-` as an added constraint, instead of a type-level `where` expression we would have:

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

But then in the Core IR we would be passing in a function's own effect as part of the type application every time we used it.

```
  y = fun %r5 !{ !Read %r1; !Console; !e1} g
```

Here's another one:

```
(==) :: 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)
```

This creates a mutability conflict, because `%r5` and `%r6` are being forced to be the same via the type variable `a`. A region can't be both Const and Mutable at the same time.

And another:

```
select :: a -> a -> a;
select x y = if ... then x else y;

pi :: Const %r2    => Float %r2;
x1 :: Mutable %r3 => Float %r3;

y  = select pi x1;
```

This seems like a reasonable thing to want to do, but creates another mutability conflict.

We don't want to be forced to copy top level constants every time we compare them with mutable data.

Bi-directional unification is not the right operation.

- Bi-directional unifiction lies at the heart of Hindley-Milner.
- It's key to designing a simple, decidable higher-order algorithm.
- It's too strong a constraint for the region/effect/closure information in these examples.

Unifying two things forces their types, regions, effects and closures to be identical.

For region/effect/closure information in the right hand side of selections we're really interested in the sum (least upper bound) of the possibilities.

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; };
```

rewrite to

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

Shape and Injection

```
(==) :: 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.

It 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.

The `Injection` constraint pushes the R/E/C info in the first argument into a l.u.b along with the second.

```
select :: (Inject a c, Inject b c)
       => a -> b -> c;

pi :: Const    %r2 => Float %r2;
x1 :: Mutable %r3 => Float %r3;

y  :: (Const %r2, Mutable %r3, Blocked %r4)
   => Float %r4
   :- %r4 = %{ %r2; %r3 }

y  = select pi x1;
```

Escape (idea)

When we suspend an application we get a suspension of type `a`.

This suspension contains a reference to the function's activation record (containing its closure `$c1`) and the argument `a`, but this isn't tracked by the type system.

```
suspend
    :: (Pure !e1, Escape a, EscapeC $c1)
    => (a -(!e1 $c1)> b) -> a -> b
```