Combinators & Staging

- Composition Function
- What Is Function
  - Ideas
  - Functions in Spaces
  - Valid Definitions Reflected in ML
    - Function `MIN`
  - What To Do with Functions (As Values!)
    - Functions Are Values!
- Staging
  - Try Currying
  - Try Staging

Composition Function

```
infix o
fun f o g = fn x => f (g x)
fun (f o g) x = f (g x) (* curried version *)
```

What Is Function

Mathematically, we consider the connections between functions, abstracting from the actual value spaces underlying the definitions.

Ideas

- Type Theory
- Category Theory

Functions in Spaces

Consider the space of **integer-valued** functions, i.e. some function taking `t` from a **type space** to some value in the **integer space**.

Then, we have **addition** of functions, capturing addition of integers.

In math, we would use a **point-wise** principal to define `f ++ g` as `(f ++ g)(x) = f(x) + g(x)`.

Here, we "lift up" the **integer space** into the space of **integer-valued functions**.

Valid Definitions Reflected in ML

```
infix ++
infix **
fun (f ++ g) x = (f x) + (g x)
fun (f ** g) x = (f x) * (g x)
```

Question. What is the type of the above functions?

```
('a -> int) * ('a -> int) -> 'a -> int
```

Note. `f` and `g` could take in different instances of `'a`, but cannot be inconsistent.

Function `MIN`

```
fun MIN (f, g) x = Int.min (f x, g x)
```

> Side Note. If `Int.min` is replaced to be polymorphic, as long as the types could be worked out, the compiler could make a `'b` type out of it.

What To Do with Functions (As Values!)

```
fun double x = 2 * x
fun square x = x * x

fun quadratic x = x * x + 2 * x

(* rewrite the above definition *)
val quadratic = square ++ double

(* functions are values *)
```

We can see the Ring structure, not upon the integer space, but now upon the integer-valued function space!

```
val lower = MIN (square, double)
(* This gives us the lower envelope of the two integer functions! *)
```

FUNCTIONS ARE VALUES!

We've **abstracted** away the explicit definition of functions. So we can treat them as **values**!

Functions Are Values!

Staging

```
fun f (x : int, y : int) : int =
  let
    val z : int = horrible_computation x
  in
    good_computation (z, y)
  end

f (5, 2) (* 10 months *)
f (5, 7) (* 10 months *)
f (5, 8) (* 10 months *)
```

We want to pull out stuff we don't want to recompute.

So consider currying!

Try Currying

```
fun g (x : int) (y : int) : int =
  let
    val z : int = horrible_computation x
  in
    good_computation (z, y)
  end

val g' = g 5  (* RT: instantaneous *)
g' 2          (* 10 months *)
g' 7          (* 10 months *)
g' 8          (* 10 months *)
```

Consider what happens when we define function `g` .

`fn x => fn y => let...end` is bound to `g` .

`fn y => let...end` and `[5/x]` is bound to `g'` .

Note. The `horrible_computation` does **not** happen when we define `g'` . Therefore, this currying does not help! So, somehow we want to do the actual computation. Here comes **staging**.

Try Staging

```
fun h (x : int) : int -> int =
  let
    val z : int = horrible_computation x
  in
    fn y => good_computation (z, y)
  end

val h' = h 5  (* 10 months *)
h' 2          (* FAST *)
h' 7          (* FAST *)
h' 8          (* FAST *)
```

Consider what happens now.

`fn x => let...in fn y =>...end` is bound to `h`.

`fn y => z + y` and `[5/x]` AND `[.../z]` is bound to `h'`.

Now this is what we want!