Jane Street

Layout Polymorphism:

Using static computation to allow efficient polymorphism over variable representations

Richard A. Eisenberg
Jane Street
reisenberg@janestreet.com

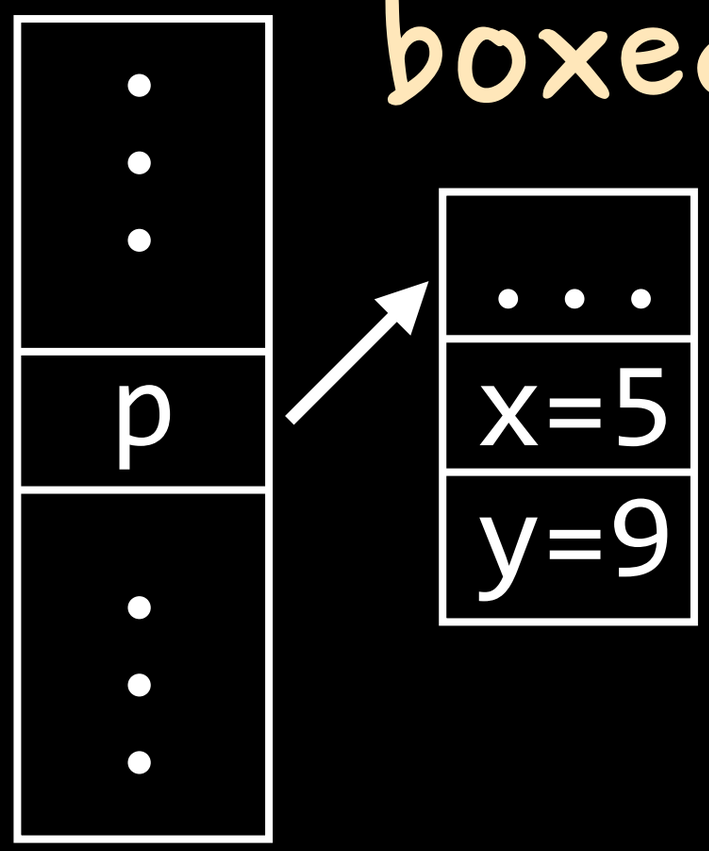
(with collaborators!)

Friday, September 8, 2023
ML Workshop
Seattle, WA, USA

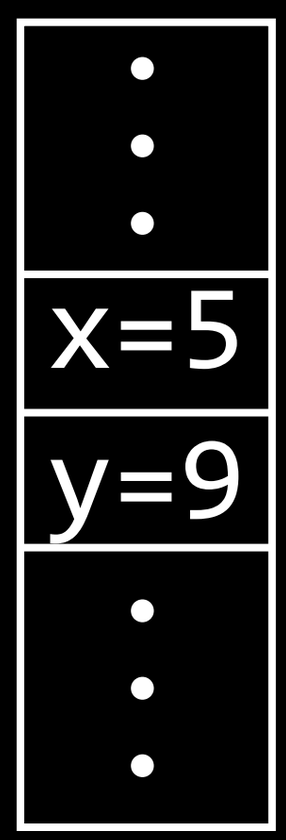
```
type point = { x : int
             ; y : int }

let p = { x = 5; y = 9 } in
```

• • •

boxed



unboxed



Unboxed types are faster than boxed ones.

Non-uniform types are harder to work with than boxed ones.

```
map : ('a -> 'b) ->
      'a array ->
      'b array
```

map must work with values of types 'a and 'b.

map must know their layouts.

memory representation
register convention

```
map : ('a -> 'b) ->
      'a array ->
      'b array
```

```
map : (int64# -> float#) ->
      int64# array ->
      float# array
```

```
map : (int64# -> float32#) ->
      int64# array ->
      float32# array
```

```
map : (int64# -> 'b) ->
      int64# array ->
      'b array
```

We need layout

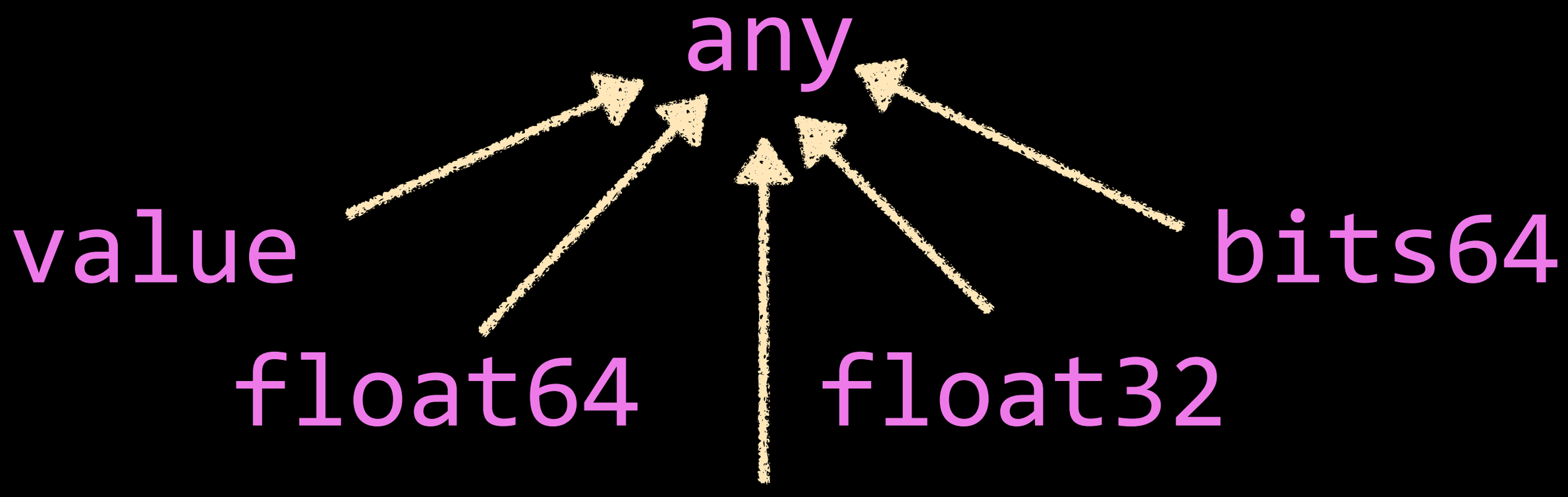
polymorphism!

flexibility!

```
map : ('a -> 'b) ->
      'a array ->
      'b array
```

```
map : ('a : any)
      ('b : any).
      ('a -> 'b) ->
      'a array ->
      'b array
```

'a and 'b can have any layout.



`value * value`

All layouts are sublayouts of `any`.

Layout polymorphism
is fine

but layout flexibility
is simpler.

How to compile

```
map : ('a : any)
      ('b : any).
      ('a -> 'b) ->
      'a array ->
      'b array
```

to efficient code?

We can't.

How to compile to efficient code?

Layout monomorphization.

Challenge:
How to monomorphize

```
map : ('a : any)
      ('b : any).
      ('a -> 'b) ->
      'a array ->
      'b array
```

translate

```
map : ('y1 : layout) ('y2 : layout).
      ('a : 'y1) ('b : 'y2).
      ('a -> 'b) ->
      'a array ->
      'b array
```

Challenge:

How to monomorphize

```
map : ('y1 : layout) ('y2 : layout).
      ('a : 'y1) ('b : 'y2).
      ('a -> 'b) ->
      'a array ->
      'b array
```

monomorphize

```
map : ('a : float64) ('b : value).
      ('a -> 'b) ->
      'a array ->
      'b array
```

We can compile that!

Challenge:
Which function?

```
M.map (f : float# -> string) arr
```

Where is the code for `map`?

It depends on `M`, a runtime `value`.

first-class module?

functor parameter?

We must know at compile time.

Challenge:
Which function?

Solution: a static mode, tracking what we can know at compile time.

Challenge:
Which function?

Substitution can happen statically.
Side effects cannot.

Preliminary typing rules in the posted extended abstract.

```
module F (X : S) = struct
  let f1 = fun ...
  let f2 = fun ...
  let map =
    if flip_coin ()
    then f1 else f2
end
```

no compile time side effects!

reject in static mode.

Challenge:
Which layouts?

Problem:
Separate compilation

Challenge:
Which layouts?

Possible solution:

Build products of layout-flexible functions include their code.
Usages induce new dependencies for the build system.

(or do C++-style deduplication)

Challenge:

m.ml Which layouts?

```
M.map (f : float# -> string) arr
```

induces a dependency on

`m_map__float64__value`

which can be produced from m.cmo

Other compilers

- MLton: whole program compilation
- C++: no abstraction; linker deduplication
- Rust: full monomorphization, not just layouts; no static mode because no modules/functors
- C/Java/Haskell: no support; just repeat the code (Haskell's type classes can help sometimes)
- C#: runtime code generation

Want to work on this?
Come visit with us for a year or two and publish research!

janestreet.com/vrp-prefaculty

Jane Street

Layout Polymorphism:

Using static computation to allow efficient polymorphism over variable representations

Richard A. Eisenberg
Jane Street
reisenberg@janestreet.com

visitor position:

Friday, September 8, 2023
ML Workshop
Seattle, WA, USA

layout-flexible:

```
map : ('a : any) ('b : any).
   ('a -> 'b) -> 'a array -> 'b array
```

layout-polymorphic:

```
map : 'y1 'y2 ('a : 'y1) ('b : 'y2).
   ('a -> 'b) -> 'a array -> 'b array
```

Polymorphism is needed only when we insist on the same layout for multiple types.