FP vs. OOP
Problem Decomposition

Two world views

FP: functions perform some operation

OOP: classes/prototypes give behavior to some kind of data

Which is better? Depends on software evolution, taste.

Each can (awkwardly) emulate the other.

Common pattern: *expressions*

Operations over type of data

Variants of a type of data

	`eval`	`toString`	`usesX`	…
`VarX`				
`Sine`				
`Times`				
…				

FP: behavior by operation

Function per operation
with branch per variant

	eval	toString	usesX	…
VarX				
Sine				
Times				
…				

Datatype with
constructor per variant

Pattern-matching selects variant.
Wildcard can merge rows in a function.

OOP: behavior by variant

Abstract base class or interface
with method per operation

	eval	toString	usesX	…
VarX				
Sine				
Times				
…				

Subclass per variant
overrides each operation method
to implement variant's behavior

Dynamic dispatch selects variant.
Concrete method in base class
can merge rows where not overridden.

FP: extensibility

	eval	toString	usesX	depth
VarX				
Sine				
Times				
Sqrt				

Add variant:
add constructor,
change all functions over datatype

Add operation:
add function,
no other changes

Static type-checker gives "to-do list"
via inexhaustive pattern-match warnings

OOP: extensibility

	eval	toString	usesX	depth
VarX				
Sine				
Times				
Sqrt				

Add variant:
add subclass
/ class implementing interface,
no other changes

Add operation:
add method
to abstract base class / interface
and all subclasses

Static type-checker gives "to-do list"
via errors about
non-overridden abstract method
/non-implemented interface method

Extensibility

Making software extensible is valuable and hard.

- If new operations likely, use FP
- If new variants likely, use OOP
- If both, use somewhat odd "design patterns"
- Reality: The future is hard to predict!

Extensibility is a double-edged sword.

- **Non-invasive reuse:** original code can be reused without changing it.
- **Difficult local reasoning/changes:** reasoning about/changing original code requires reasoning about/changing remote extensions.

Restricting extensibility is valuable.

- ML abstract types
- Java `final`

Binary Operations

What about operations that take two arguments of possibly different variants?

- Include value variants `Int`, `Rational`, `...`
- (Re)define `Add` to work on any pair of `Int`, `Rational`, `...`

The addition operation alone is now a *different* 2D grid:

Add	Int	Rational	...
Int			
Rational			
...			

ML approach: pattern-matching

Natural: pattern-match both simultaneously

```
fun add_values (v1,v2) =
  case (v1,v2) of
     (Int i, Int j) => Int (i+j)
   | (Int i, Rational(n,d)) => Rational (i*d+n,d)
   | (Rational _, Int _) => add_values (v2,v1)
   | ...

fun eval e =
  case e of
     ...
   | Add(e1,e2) => add_values (eval e1, eval e2)
```

OOP approach: dynamic dispatch

```
abstract class Value extends Expr {
  ...
  Value addValues(Value v);
}

class Add extends Expr {
  ...
  Value eval() {
    e1.eval().addValues(e2.eval())
  }
}

class MyInt extends Value {
  …
  // add this to v
  Value addValues(Value v) {
    … // what goes here?
  }
}
```

Dynamic dispatch chooses `addValues` based on result of `e1.eval()`

Depends on what kind of value v is.

Explicit *Double Dispatch*

OOP: Make variant choices using dynamic dispatch.

```
abstract class Value extends Expr {
  Value addValues(Value v);
  Value addInt(MyInt v);
  Value addRational(MyRational v);
}

class MyInt extends Value {
  ...
  Value addValues(Value) { return v.addInt(this); }
```

Dynamic dispatch on first value got us here.

Now, dispatch on second value, "telling it" what kind of value `this` is.

```
  Value addInt(MyInt v) { ... }
  Value addRational(MyRational v) { ... }
}
```

Repeat for all Value subclasses...

Reflecting

Double dispatch manually emulates basic pattern-matching.

- An analogous FP pattern emulates dynamic dispatch.

Does it change the way in which OOP handles evolution?

- Add an operation over pairs of Values:
  - OOP double dispatch: how many added / changed classes?
  - FP pattern matching: how many added / changed functions?

- Add a kind of Value:
  - OOP double dispatch: how many added / changed classes?
  - FP pattern matching: how many added / changed functions?

What if we could dispatch based on all arguments at once?

Multiple dispatch / multimethods

Dynamic dispatch on *all arguments*.

- One version of method per combination of argument types.
- NOT static overloading.
- Remarkably close to functions that pattern-match arguments.
  - But the individual branches may be split up.
  - But subtyping can lead to ambiguous dispatch.

If dynamic dispatch is essence of OOP, multiple dispatch is its natural conclusion.

Old research idea picked up in some recent languages (e.g., Clojure, Julia)

Closures vs. Objects

Closure:

- Captures code of function, by function definition.
- Captures all bindings the code may use, by lexical scope of definition.

Object:

- Captures code for all methods that could be called on it, by class hierarchy.
- Captures bindings that may be used by that code, by instance variables declared in class hierarchy.

Each can (awkwardly) emulate the other.