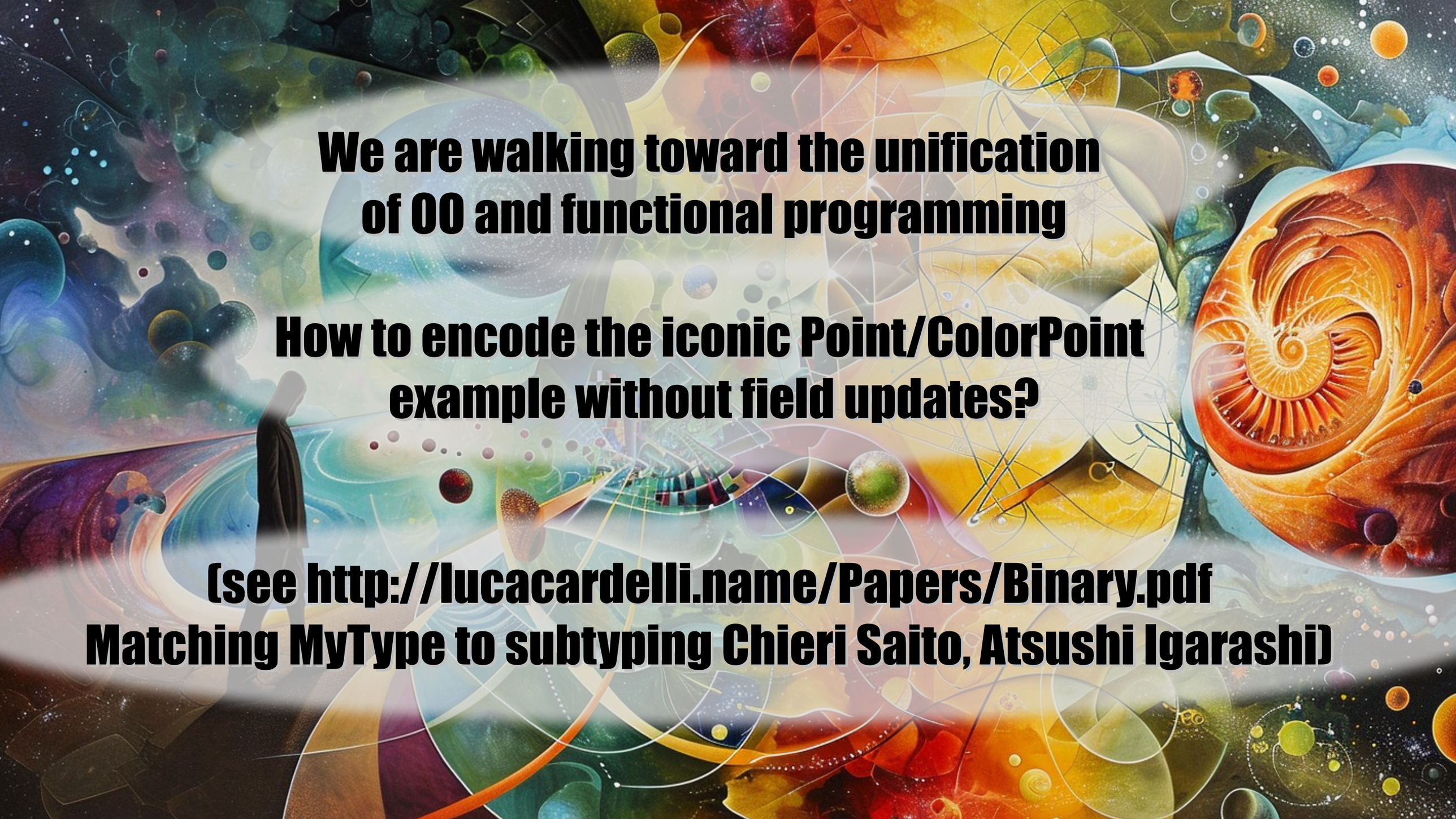
Marco Servetto

Victoria University of Wellington

Withers and the unification of OOP and FP

(work in progress)


We are walking toward the unification
of OO and functional programming
How to encode the iconic Point/ColorPoint
example without field updates?
(see http://lucacardelli.name/Papers/Binary.pdf
Matching MyType to subtyping Chieri Saito, Atsushi Igarashi)

I’m starting to think this is like the expression problem because:

- many possible solutions
- all kind of unsatisfactory
- different kinds of methods may surprisingly not be supported

Set up:

```
interface Point{
  int x();
  int y();
  Point withX(int x);
  Point withY(int y);
  String id(); //example method showing overriding on various kinds of points
  default Point times(int r){return this
    .withX( this.x() * r )   //example method body using withers to return a 'Point'
    .withY( this.y() * r );  //We want not to duplicate this body in 'CPoint'
  }
  default Point max(int x){//example method body using withers and occasionally
    if ( this.x() > x ){ return this; }//returning 'this'
    return this.withX(x);
    }
  }
record Point1(int x, int y) implements Point{//example implementations of points
  public Point1 withX(int x){ return new Point1(x,this.y()); }//ideally, withX/Y could
  public Point1 withY(int y){ return new Point1(this.x(),y); }//be auto-generated
  public String id(){ return "Point1"; }
}
record Point2(int x, int y) implements Point{//We could also use anonymous classes
  public Point2 withX(int x){ return new Point2(x,this.y()); }//!
  public Point2 withY(int y){ return new Point2(this.x(),y); }//!
  public String id(){ return "Point2"; }
}
```

Set up:

```
interface CPoint extends Point{
  String color();
  CPoint withColor(String c);
  CPoint withX(int x);//return type refinement
  CPoint withY(int y);//could it be autogenerated? No record here

}
record CPoint1(int x, int y,String color) implements CPoint{
  public CPoint withX(int x){ return new CPoint1(x,this.y(),this.color()); }//!
  public CPoint withY(int y){ return new CPoint1(this.x(),y,this.color()); }//!
  public CPoint withColor(String c){ return new CPoint1(this.x(),this.y(),color); }//!
  public String id(){ return "CPoint1"; }
}
```

Set up:

```
interface CPoint extends Point{
  String color();
  CPoint withColor(String c);
  CPoint withX(int x);//return type refinement
  CPoint withY(int y);//could it be autogenerated? No record here

}
record CPoint1(int x, int y,String color) implements CPoint{
  public CPoint withX(int x){ return new CPoint1(x,this.y(),this.color()); }//!
  public CPoint withY(int y){ return new CPoint1(this.x(),y,this.color()); }//!
  public CPoint withColor(String c){ return new CPoint1(this.x(),this.y(),color); }//!
  public String id(){ return "CPoint1"; }
}

public class Example {
  public static void main(String[] a){
    Point p= new Point1(1, 2);//or Point2
    p = p.withX(0);
    p = p.times(5);//all good
    CPoint cp= new CPoint1(1, 2, "red");
    cp = cp.withX(3);//Works, but directly overridden
    cp = cp.times(4);//Type error: 'times' returns a Point
  }    //We can not do code reuse involving calling the withers.
}      //But with setters, code reuse would be no problem. Is this a limitation of FP?
```

Idea: use generics

```
interface Point<S>{
  int x();
  int y();
  S withX(int x);
  S withY(int y);

  String id();

  default S times(int r){return this
    .withX(this.x()*r)//ok
    .withY(this.y()*r);//does S has a withY?
  }
  default S max(int x){
    if(this.x()>x){ return this; }
    return this.withX(x);
  }
}

record Point1(int x, int y) implements Point<Point1>{
  public Point1 withX(int x){ return new Point1(x,this.y()); }//!
  public Point1 withY(int y){ return new Point1(this.x(),y); }//!
  public String id(){ return "Point1"; }
}
```

Idea: use generics

```
interface Point<S extends Point<S>>{
  int x();
  int y();
  S withX(int x);
  S withY(int y);

  String id();

  default S times(int r){return this
    .withX(this.x()*r)//ok
    .withY(this.y()*r);//does S has a withY? Now it does!
  }
  default S max(int x){
    if(this.x()>x){ return this; }
    return this.withX(x);
  }
}

record Point1(int x, int y) implements Point<Point1>{
  public Point1 withX(int x){ return new Point1(x,this.y()); }//!
  public Point1 withY(int y){ return new Point1(this.x(),y); }//!
  public String id(){ return "Point1"; }
}
```

Idea: use generics

```
interface Point<S extends Point<S>>{
  int x();
  int y();
  S withX(int x);
  S withY(int y);

  String id();

  default S times(int r){return this
    .withX(this.x()*r)
    .withY(this.y()*r);
  }
  default S max(int x){
    if(this.x()>x){ return this; }//Can you spot the error here?
    return this.withX(x);
  }
}

record Point1(int x, int y) implements Point<Point1>{
  public Point1 withX(int x){ return new Point1(x,this.y()); }//!
  public Point1 withY(int y){ return new Point1(this.x(),y); }//!
  public String id(){ return "Point1"; }
}
```

Idea: use generics

```
interface Point<S extends Point<S>>{
  int x();
  int y();
  S withX(int x);
  S withY(int y);

  String id();

  default S times(int r){return this
    .withX(this.x()*r)
    .withY(this.y()*r);
  }
  default S max(int x){
    if(this.x()>x){ return this; }//Can you spot the error here?
    return this.withX(x);//we can not use Point<S> instead of S in the results,
  } //we need S in times/max to be able to reuse its body for both Point/CPoint
}

record Point1(int x, int y) implements Point<Point1>{
  public Point1 withX(int x){ return new Point1(x,this.y()); }//!
  public Point1 withY(int y){ return new Point1(this.x(),y); }//!
  public String id(){ return "Point1"; }
}
```

Idea: use generics

```
interface Point<S extends Point<S>>{
  int x();
  int y();
  S withX(int x);
  S withY(int y);
  S self();
  String id();

  default S times(int r){return this
    .withX(this.x()*r)
    .withY(this.y()*r);
  }
  default S max(int x){ //the 'self' method solves this.
    if(this.x()>x){ return this.self(); }
    return this.withX(x);//we can not use Point<S> instead of S in the results,
  } //we need S in times/max to be able to reuse its body for both Point/CPoint
}

record Point1(int x, int y) implements Point<Point1>{
  public Point1 self(){ return this; }//!
  public Point1 withX(int x){ return new Point1(x,this.y()); }//!
  public Point1 withY(int y){ return new Point1(this.x(),y); }//!
  public String id(){ return "Point1"; }
}
```

Usage

```
interface Point<S extends Point<S>>{
  ...x/y/withX(int x)/withY(int y)/self/id
  default S times(int r){...}
  default S max(int x){...}
}
record Point1(int x, int y) implements Point<Point1>{
  ..self,withX,withY//!
  public String id(){ return "Point1"; }
}
interface CPoint<S extends CPoint<S>> extends Point<S>{
  String color();
  S withColor(String c);
}
record CPoint1(int x, int y,String color) implements Cpoint<CPoint1>{
  ..self,withX,withY,withColor//!
  public String id(){ return "CPoint1"; }
}
...
  public static void main(String[] a){
    Point<?> p= new Point1(1, 2);//Ugly, all types use <?> now
    p = p.withX(0);
    p = p.times(5);
    CPoint<?> cp= new CPoint1(1, 2, "red");
    cp = cp.withX(3);
    cp = cp.times(4);//But it works!
  }
```

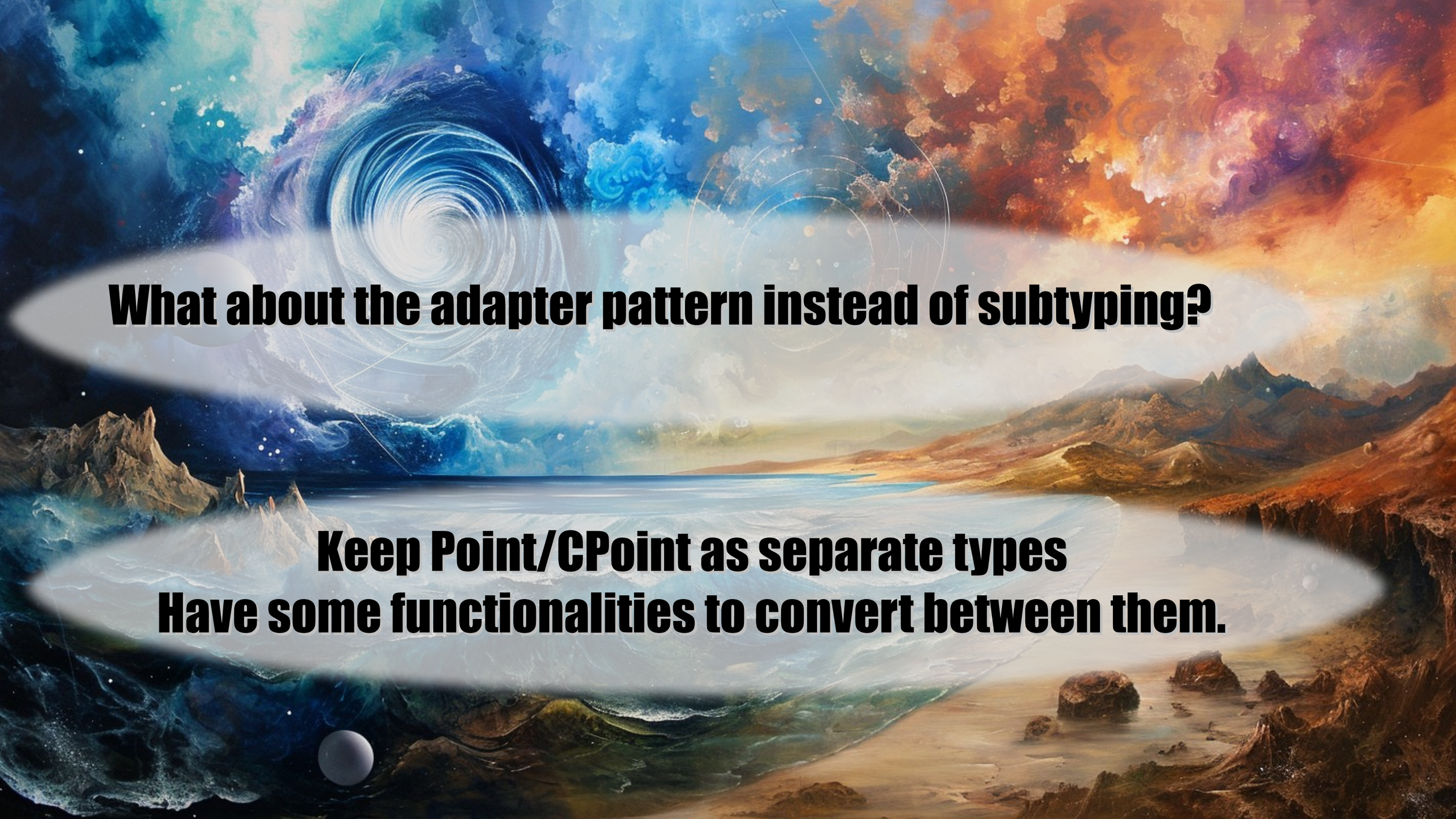
Extension: Diff method

```
interface Point<S extends Point<S>>{
...x/y/withX(int x)/withY(int y)/self/id
  default S times(int r){...}
  default S max(int x){…}
  default String diff(S other){
    return "(" + this.x() + "/" + other.x() + "," + this.y() + "/" + other.y() + ")";
  }
}
interface CPoint<S extends CPoint<S>> extends Point<S>{
  String color();
  S withColor(String c);
  default String diff(S other){//Overridden to print the full difference
    return "(" + this.x() + "/" + other.x() + "," + this.y() + "/" + other.y() + ","
      + this.color() + "/" + other.color() + ")";
  }
}
...
  public static void main(String[] a){
    Point<?> p= new Point1(1, 2);
    CPoint<?> cp= new CPoint1(1, 2, "red");
    capture(p);//can call diff by using wildcard capture!
    capture(cp);
  }
  public static <X extends Point<X>>
  void capture(Point<X> p){
    Point<X> p1= p.withX(10);
    System.out.println(p.diff(p1.self()));
  }
```

This works, but it uses the most sophisticated features of the language.

I argue that if the functional Point / Color Point example is much harder to teach that the mutable version, we do not have a good unification of OOP and FP

We could make it ‘look good’ superficially, by auto-generating many things, and assuming a first generic parameter on everything with the right shape, so we can write ‘Point’ but we mean Point<?> logically. Would this be a solution, or is it just hiding the dust?


What about the adapter pattern instead of subtyping?
Keep Point/CPoint as separate types
Have some functionalities to convert between them.

```
interface Point{
  int x();  int y();  Point withX(int x);  Point withY(int y);
  String id();
  default Point times(int r){ return this.withX(this.x()*r).withY(this.y()*r); }
  default Point max(int x){
    if(this.x()>x){ return this; }
    return this.withX(x);
  }
}
```

```
interface Point{
  int x();  int y();  Point withX(int x);  Point withY(int y);
  String id();
  default Point times(int r){ return this.withX(this.x()*r).withY(this.y()*r); }
  default Point max(int x){
    if(this.x()>x){ return this; }
    return this.withX(x);
  }
}
interface CPoint{//Note: does not implement Point
  int x();  int y();  String color();
  CPoint withX(int x);  CPoint withY(int y);  CPoint withColor(String c);
  String id();
  default CPoint withPoint(Point p){ return this.withX(p.x()).withY(p.y()); }
  default Point toPoint(){..}

  default CPoint times(int r){
    return this.withPoint(this.toPoint().times(r)); //first attempt
  }
  default CPoint max(int x){
    return this.withPoint(this.toPoint().max(x));//first attempt
  }
}
```

```
interface Point{
  int x();  int y();  Point withX(int x);  Point withY(int y);
  String id();
  default Point times(int r){ return this.withX(this.x()*r).withY(this.y()*r); }
  default Point max(int x){
    if(this.x()>x){ return this; }
    return this.withX(x);
  }
}
interface CPoint{//..
  String id();
  default CPoint times(int r){ return this.withPoint(this.toPoint().times(r)); }
  default CPoint max(int x){ return this.withPoint(this.toPoint().max(x)); }

  default CPoint withPoint(Point p){ return this.withX(p.x()).withY(p.y()); }
  default Point toPoint(){ return new Point1(this.x(), this.y()); }
}//returning a Point1 every time is wrong.
 //asking every Cpoint to override toPoint also looks wrong.
```

```
interface Point{..
  default Point times(int r){ return this.withX(this.x()*r).withY(this.y()*r); }
  ..}

interface CPoint{..
  default CPoint times(int r){ return this.withPoint(this.toPoint().times(r)); }

  default CPoint withPoint(Point p){ return this.withX(p.x()).withY(p.y()); }

  default Point toPoint(){//We can use the adapter pattern!
    CPoint self = this;
    return new Point(){
      public int x(){ return self.x(); }
      public int y(){ return self.y(); }
      public Point withX(int x){ return self.withX(x).toPoint(); }
      public Point withY(int y){ return self.withY(y).toPoint(); }
      public String id(){ return self.id(); }
      public Point times(int r){ return self.times(r).toPoint(); }
      public Point max(int x){ return self.max(x).toPoint(); }
    };
  }
}
//Can you spot the problem?
//stack overflow times → toPoint → times..
```

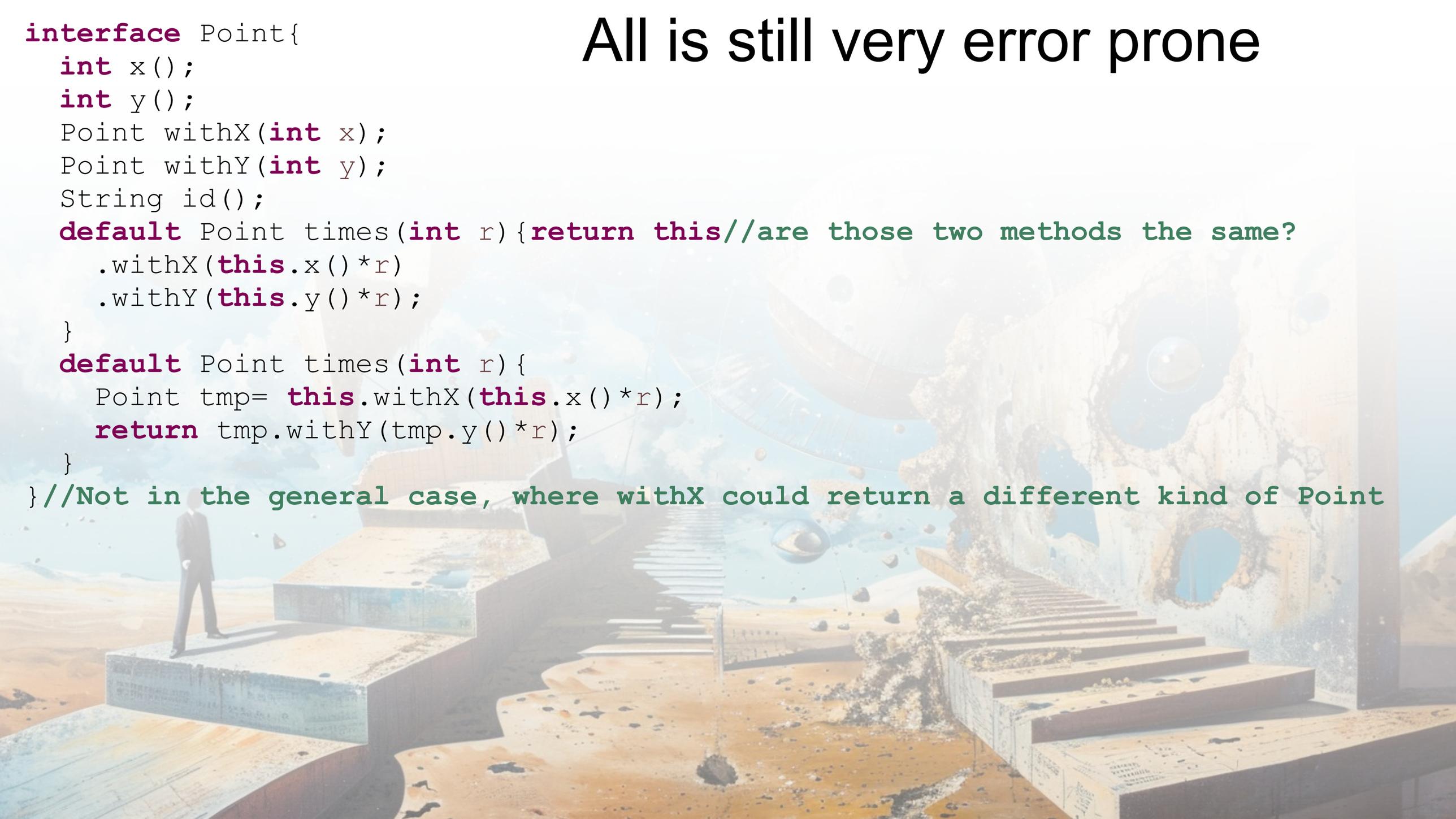
```
interface Point{..
  default Point times(int r){ return this.withX(this.x()*r).withY(this.y()*r); }
  ..}

interface CPoint{..
  default CPoint times(int r){ return this.withPoint(this.toPoint().times(r)); }

  default CPoint withPoint(Point p){ return this.withX(p.x()).withY(p.y()); }
//withPoint is also wrong since it is keeping the 'id' behavior of the initial CPoint
//not the 'id' behavior of the result of 'times'
//What if there is a special kind of CPoint that overrides times to return a CPoint with
//a different 'id'?
```

```
interface Point{//Interlocked adapter pattern?
  int x(); int y();   Point withX(int x);  Point withY(int y);
  String id();
  default Point times(int r){ return this.withX(this.x()*r).withY(this.y()*r); }
  default Point max(int x){ .. }
  default CPoint toCPoint(CPoint other){
    Point self = this;
    return new CPoint(){
      public int x(){ return self.x(); }
      public int y(){ return self.y(); }
      public CPoint withX(int x){ return self.withX(x).toCPoint(other); }
      public CPoint withY(int y){ return self.withY(y).toCPoint(other); }
      public String color(){return other.color();}
      public CPoint withColor(String c){ return self.toCPoint(other.withColor(c)); }
      public String id(){ return self.id(); }
      public CPoint times(int r){ return self.times(r).toCPoint(other); }
      public CPoint max(int x){ return self.max(x).toCPoint(other); }
      };}
  }
}
interface CPoint{..
  default Point toPoint(){//Here toCPoint does the unwrapping
    CPoint self = this;
    return new Point(){..
      public CPoint toCPoint(CPoint other){ return self; }
    };
  }
```


Can you tell I'm growing desperate?

All is still very error prone

```
interface Point{
  int x();
  int y();
  Point withX(int x);
  Point withY(int y);
  String id();
  default Point times(int r){return this//are those two methods the same?
    .withX(this.x()*r)
    .withY(this.y()*r);
  }
  default Point times(int r){
    Point tmp= this.withX(this.x()*r);
    return tmp.withY(tmp.y()*r);
  }
}//Not in the general case, where withX could return a different kind of Point
```



What is the guarantee of ‘withX/times’?

```
interface Point{
  int x();
  int y();
  Point withX(int x);
  Point withY(int y);
  String id();
  default Point times(int r){return this//are those two methods the same?
    .withX(this.x()*r)//on field update, we keep the ‘dynamic type’ of this
    .withY(this.y()*r);//Do we? What about python
  }
  default Point times(int r){
    Point tmp= this.withX(this.x()*r);
    return tmp.withY(tmp.y()*r);
  }
}//Not in the general case, where withX could return a different kind of Point
```

Other ideas/options

- Super expressive structural types (co-induction)
  - Not my strong suite
- SelfType/This/Mytype/Self
  - This avoid the times example error before. But… is this killing oo?
- Value based languages
  - Another version where the ‘withX’ is statically guaranteed to return the same dynamic type of the receiver instead of allowing for subtypes.

The ‘?’ solution is not ‘MyType’

```
interface Point<S extends Point<S>>{
  int x();  int y();  S withX(int x);  S withY(int y);  S self();
  String id();
  default S times(int r){ return this.withX(this.x()*r).withY(this.y()*r); }
  default S max(int x){..}
  default String diff(S other){
    return "("+this.x()+"/"+other.x()+","+this.y()+"/"+other.y()+")";
  }
}
record Point2(int x, int y) implements Point<Point2>{
  public Point2 self(){ return this; }
  public Point2 withX(int x){ return new Point2(x,this.y()); }
  public Point2 withY(int y){ return new Point2(this.x(),y); }
  public String id(){ return "Point2"; }
}
record Point3(int x, int y) implements Point<Point2>{
  public Point2 self(){ return new Point2(x,y); }
  public Point2 withX(int x){ return new Point2(x,this.y()); }
  public Point2 withY(int y){ return new Point2(this.x(),y); }
  public String id(){ return "Point3"; }
}
```

A situation where mutation GIVES a GUARANTEE

Point p=..

At a certain moment, p is exactly a Point1

in the void setters there is no way the type is changed

in the withers, now there is a way to change the type

Point<?> p=..

Point<?> p2=p.withX(3);

The big choice:

- Withers are statically enforced to return the exact type of the receiver (in the sense of dynamic dispatch behaving the same)
  - There will be methods ‘using withers’ that also are statically enforced to return the same exact type
- Withers can return a subtype as usual in any other OO methods
  - More consistent/simpler behavior?

Two forms of subtyping in OO?

- Subtyping as dynamic dispatch can differ between different types of Points
  - This is an extension of FP first class functions: the behavior of calling an int→str function is dynamically dispatched
- Subtyping as feature growth, as between Point and ColorPoint
  - This is more connected with record subtyping

Questions / Discussion