Introduction to Object Oriented Programming

A. N. Clark.

1. Traditional View Of Data & Operations

Languages like Pascal & Algol have data structures & operations defined separately

eg

```
struct point        <- type name
    x : int ;
    y : int ;       } fields available to any part of the program
end

function addpts (p1: point, p2: point)
    new point (p1.x + p2.x , p1.y + p2.y)
end
```

Disadvantages

- changes to the data structures, eg. point, require changes to all parts of the program which use it.
  eg changing the name of a field
- the purpose of the data structure is not evident
  cannot determine from the definition of the structure which operations may be performed on it.
  operation may be defined which conflict with the original intention.
  not even obvious where the operations are defined since they are distributed around the body of the program.
- this open approach cannot easily support the reuse of code since changes to the definition have non local scope.

2. Data Abstraction

Define the representation of the data structure and the operations which are defined to manipulate it in one place.

"Abstract the operations into a public interface".

eg **type** point **is**

```
    x : int;
    y : int;
```

} Representation to be thought of as private.

Operations which may be performed on a point to be thought of as public.

```
    getx (p: point) = p.x ;
    gety (p: point) = p.y ;
    add (p1: point) (p2: point) =
            new point ( getx p1 + getx p2 ,
                        gety p1 + gety p2 )
end
```

Other parts of the program can only make new point structures or manipulate point structures through the defined interface.

The representation is hidden. Changes to the representation have local scope and will not affect other parts of the program. The definition is <u>encapsulated</u>.

It is obvious what the intention of the data structure is — can reason about the structures in terms of the operations.

3 Interfaces

Every data value can be tagged with the type.

Each operation must execute with respect to the type (because an operation may be found in more than one interface).

Operations <u>dispatch</u> on the type of the argument.

Example:

A complex number has an interface

```
real-part
imag-part
magnitude
angle
```

} public

A complex number may be represented in rectangular or polar format.

But a <u>client</u> (using) program does not care what

format the structure is in, providing that the interface is supported.

```
let polar = newtag
    polar-real-part  = λp. (p.r) * (cos p.a)
    polar-imag-part  = λp. (p.r) * (sin p.a)
    polar-magnitude  = λp. p.r
    polar-angle      = λp. p.a
in
    let make-polar (r, a) = seal polar {r = r ; a = a}

        real-part (p: polar) = polar-real-part p
        imag-part (p: polar) = polar-imag-part p
        magnitude (p: polar) = polar-magnitude p
        angle     (p: polar) = polar-angle p
```

The operators in the interface are overloaded for polar argument.

```
let rectangular = newtag
    rect-real-part  = λr. r.x
    rect-imag-part  = λr. r.y
    rect-magnitude  = λr. √(r.x² + r.y²)
    rect-angle      = λr. arctan (r.y ; r.x)
in
    let make-rectangular (x,y) = seal rectangular {x = x ; y = y}

        real-part (p: rectangular) = rect-real-part p
        imag-part (p: rectangular) = rect-imag-part p
        magnitude (p: rectangular) = rect-magnitude p
        angle     (p: rectangular) = rect-angle p
```

Advantages

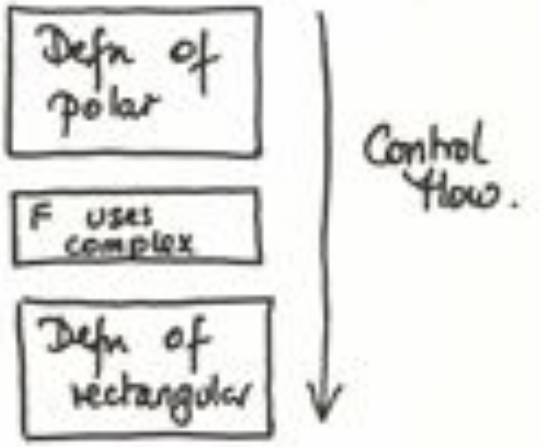
- Encapsulation
- private part of the representation may be defined used only by the operations
- private operations may be defined, not part of the public interface.

Disadvantages

- Each operation must be overloaded separately.
- The type of an argument to an operation must be known at the point of use in order that dispatching can be resolved.
- Client is active partner
  ds is passive partner

4 Problems associated with Overloading

Scenario




```
let polar = newtag
    ⋮
in
  let make-polar ...
      angle (p: polar) = polar-angle p
  in
    let F = λc. angle c
    in
      let rectangular = newtag
          ⋮
      in
        let make-rectangular ...
            angle (p: rectangular) = rectangle p
        in
          let c = make-rectangular (1φ, 2φ)
          in
            F c        ERROR.
```

To see why, think of overloading as redefining the function.

(i) All functions exist, before they are defined by the programmer, as

```
let fn = λx. error "Undefined function fn"
```

(ii) An overloading for a datatype t, to perform tfn

```
let fn① = λo. let tag = Shiptag o
              in
                 if tag = t
                 then tfn o
                 else fn②
in
  ...
```

① = new definition of fn for rest of scope

② = previous definition of fn inc. any overloading.

So when angle is overloaded,

```
let angle = λp. let tag = Shiptag p
                in
                   if tag = polar
                   then polar-angle p
                   else (λo. error "undefined function angle")p
in
   let F = λc. angle c
   in
      let angle = λp. let tag = Shiptag p
                      in
                         if tag = rectangular
                         then rect-angle p
                         else angle p
in
   let c = Make-rectangular (1φ, 2φ)
   in
      F c
```

This is a disadvantage because F is defined in terms of complex numbers – <u>not</u> polar & rectangular.

5. Discrimination Functions

In order to allow the same function to be overloaded with respect to all the implementations of complex numbers and possibly other data types, the function may be declared to discriminate between the types.

```
let angle = λo. case structag o of
                    rectangular : rect-angle o
                    polar       : polar-angle o
                    otherwise   : error "Cannot discriminate"
                 end
```

But this is no good since all the data types which angle will have to deal with must be known in advance.

have to deal with must be known in advance.

Adding a new type with angle in its interface will require a change to the definition of angle.

Making Values the active partner

Suppose that a list is to be represented as

cons(h,t) or nil

Using overloading,

```
List : union
           cons(h,t)
           nil
       end
```

```
head : cons(h,t) = h
       nil       = error
```

```
tail : cons(h,t) = t
       nil       = error
```

Now want to introduce a new data value of type list which is nil*. The head and tail of this data value in itself.

All of the boxes must be changed!

list : <u>unions</u>
cons(h,t)
nil
nil*
<u>end</u>

head :	cons(h,t) = h
	nil = error
	nil* = nil*

tail :	cons(h,t) = t
	nil = error
	nil* = nil*

If the data values are allowed to handle the operations then adding a new data value is easy.

list :	head
	tail

cons(h,t) :	head = h
	tail = t

nil :	head = error
	tail = error

Now add nil*

nil* :	head = nil*
	tail = nil*

NB easy to add new values but difficult to add new operations

– problem has been inverted

but – perhaps inheritance (see later) can help.

7. Objects (and a whole lot more!)

Some of the problems which have been outlined above can be solved by a technique called object oriented programming.

The idea is that data and code merge into an entity called an object. Objects may contain other objects ~~and~~ but all objects are sealed from the outside world. All you can do to an object is request that it performs an operation.

Programs are written as a series of object definitions, and an initial configuration of objects. A program is initialised by requesting that one object from the initial configuration perform a task.

An object is a data value with,

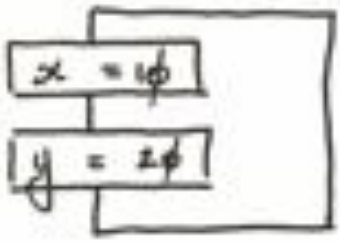
- its own local state
- an identity which is pervasive between tasks (this is ignored but it is important to people involved in DB's and IIRM).
- its own operations which have exclusive access to the local state. ①
- an interface of operations, a subset of ① which are available to clients.

- A client A of an object B is another object which
  - has created B
  - has received B as part of information passed as an actual parameter
  - contains B as part of its local state
  - has computed B by a return value from another object, C, which A is a client of.

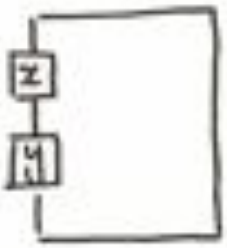
Given an expression E in a box diagram and a free variable i in E, the search for the place where i is bound starts in the box which surrounds E and continues outward.

If no binding for i is found, ie the outermost box is reached then i is not bound in E.

A record $\{x = 1\phi ; y = 2\phi\}$ can be represented as,

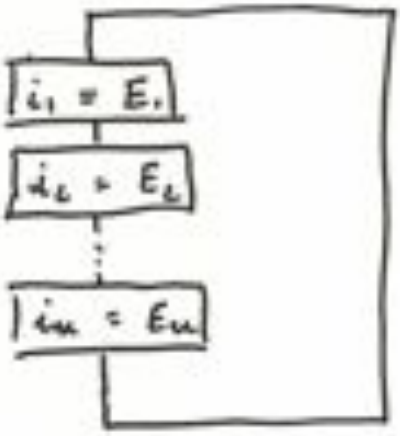


or



if the values are unimportant.

In general a record will be,

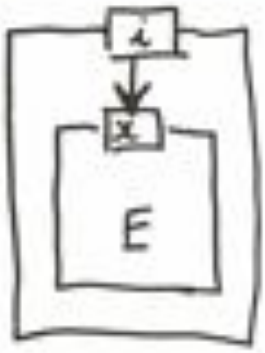


Given a free variable i in E_j to find the place where it is bound:

is it i_{j-1} ? no ↓
is it i_{j-2} ? no ↓
⋮
is it i_1 ? no ↓
try the surrounding box.

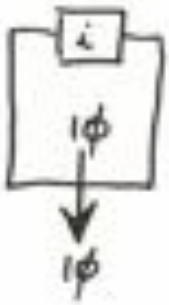
To show the flow of data - arrows are used,

$\lambda i. (\lambda x. E) i$ would be represented as,



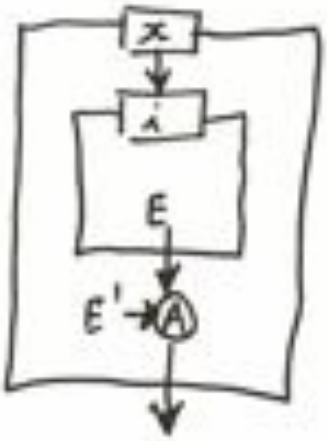
The result of evaluating an expression E will be fed out of the back end of the box which contains it.

$\lambda i.\ 1\phi$

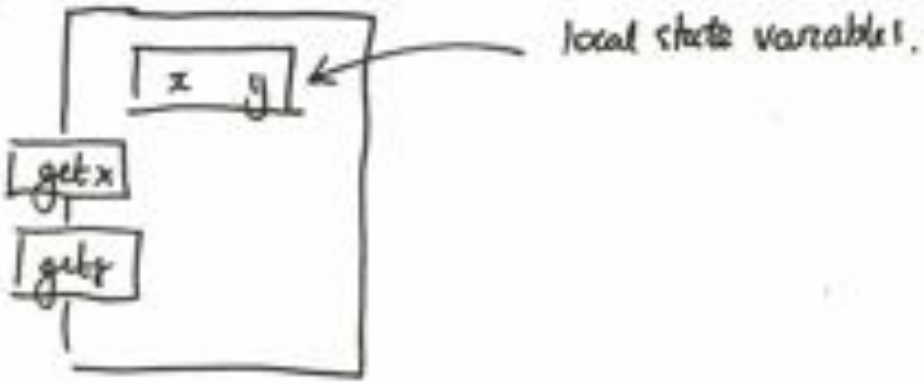


NB this is a <u>static</u> model which does not seem to deal with the following very well,

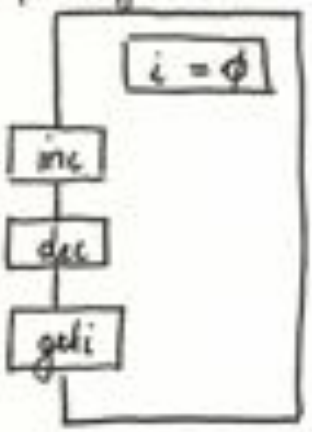
$$\lambda x.\ ((\lambda i.E)\ x)\ E'$$



Records will be used to represent objects. The local state identifiers will be made explicit, by placing their names in a box at the top of the record. This is purely for annotation and does not affect binding scope in any way.



9. An Example Object



A Counter Object

Performing the inc operation will return a new counter object which has a value $i+1$

Performing the dec operation will return a new counter object which has a value $i-1$

Performing the geti operation will return the value of i.

10. The Message Passing Metaphor

The operations which are made available to client objects are called <u>methods</u>.

Executing a method involves a <u>client</u> object requesting that the <u>target</u> object apply the function to an actual parameter supplied by the client object.
This is known as <u>message passing</u>.

Message passing differs from ordinary function application because the method is held in the target object. No overloading is involved.

Consider the counter object,

```
let rec make-counter i =
        { inc = λ().  make-counter (i+1) ;
          dec = λ().  make-counter (i-1) ;
          geti = λ().  i }
in
    let  o =  make-counter ϕ
```

(o. inc ()).geti () will return 1

((o.dec()). dec()).geti will return -1

11 Why Message Passing?

- Overloading requires that the data type be known.
- Message passing relieves the client of the responsibility of knowing which operator to apply.
- In some cases the client cannot know the type of the object because it is defined at a late time in the program.
- Providing that the target contains the operation in its interface the client does not need to know its type.

NB this is not quite true since the operation may be available but not perform as expected. Behavioral constraints need to be applied.

12. Generators

Objects are created by functions called <u>generators</u>.

The generator takes some initialization parameters and returns the object.

Some object oriented languages call the generator the <u>class</u> although this may contain more information than is specified here.

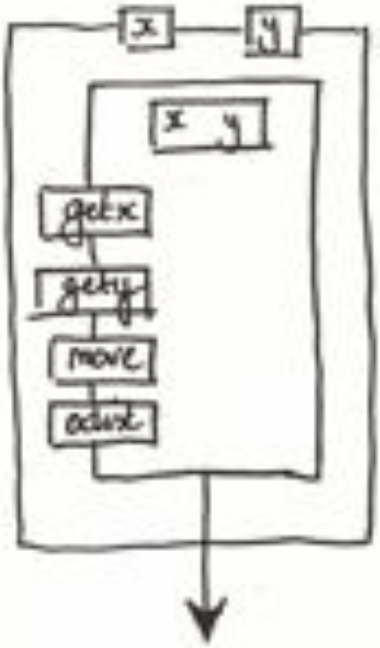
The generator will be used within the methods of an object to effect local state change. So a new object is returned with the new values.

A point in 2D Space

<u>let</u> <u>rec</u> make-point (x, y) =

$$\{ \begin{array}{l} \text{get}x = \lambda().x; \\ \text{get}y = \lambda().y; \\ \text{move} = \lambda\delta x.\ \lambda\delta y.\ \text{make-point}\ (x+\delta x, y+\delta y) \\ \text{adist} = \lambda().\ \sqrt{x^2+y^2} \end{array} \}$$

The point object generator looks like :



Now suppose that a box generator is required,

<u>let</u> <u>rec</u> make-box (x, y, l, w) =

$\{$ getx = $\lambda()$. x;
getY = $\lambda()$. y;
getl = $\lambda()$. l;
getw = $\lambda()$. w;
move = $\lambda\delta x$. $\lambda\delta y$. make-box $(x+\delta x, y+\delta y, l, w)$;
resize = $\lambda\delta l$, $\lambda\delta w$. make-box $(x, y, \delta l+l, \delta w+w)$
edist = $\lambda()$. $\sqrt{x^2+y^2}$ $\}$

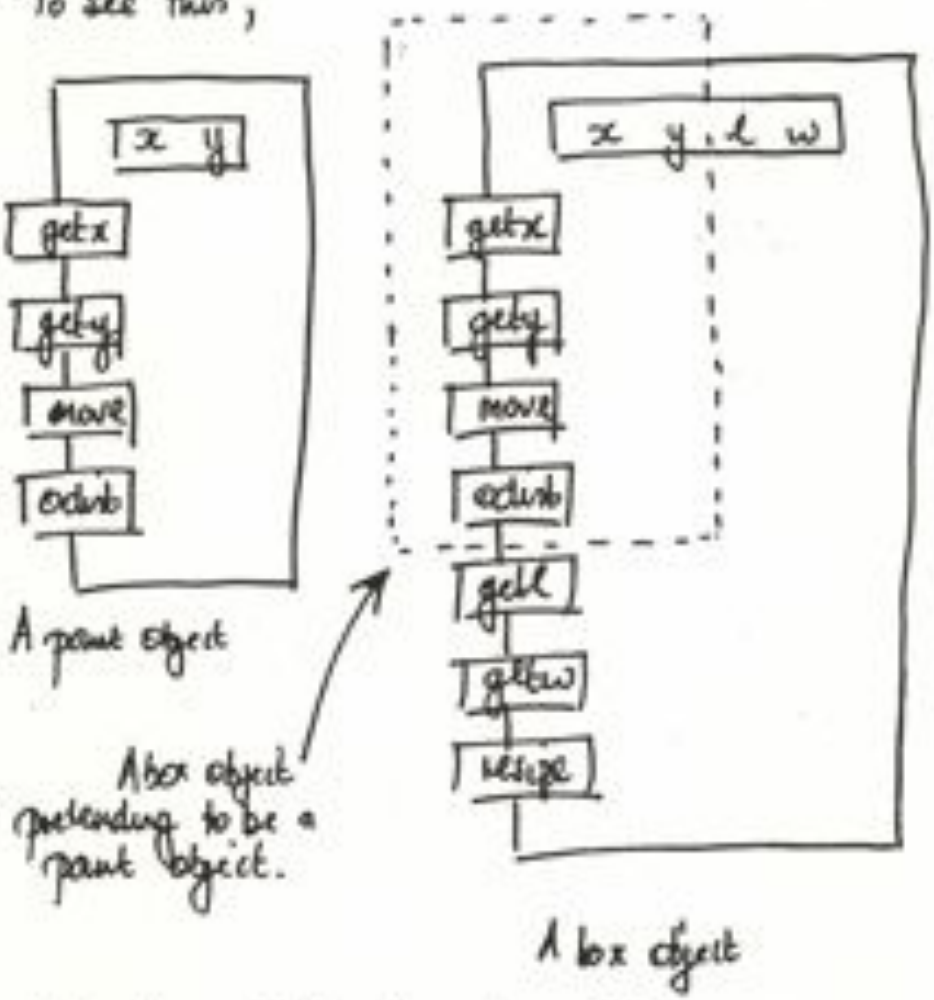
But the generator for a box object is very like that for a point object.

The box generator extends the point generator by adding a few more methods and local state identifiers.

This type of extension is called <u>horizontal extension</u>.

Any program which manipulates a point object will accept a box object and never know the difference. Since it was defined to use a point object it will never make use of the extra state or methods.

To see this,



Since the methods of a box object are a superset of those of a point object how can they be used in the box generator?

13. The ⊕ operator

The ⊕ operator is a binary infix op which takes records as arguments and concatenates them producing a new record.

if $\underline{\text{fields}}\ r1 = \{i_1, \ldots i_n\}$

$\underline{\text{fields}}\ r2 = \{i'_1, \ldots, i'_m\} \cap \underline{\text{fields}}\ r1 = \phi$

then

$\underline{\text{fields}}\ (r1 \oplus r2) = \underline{\text{fields}}\ r1 \cup \underline{\text{fields}}\ r2$

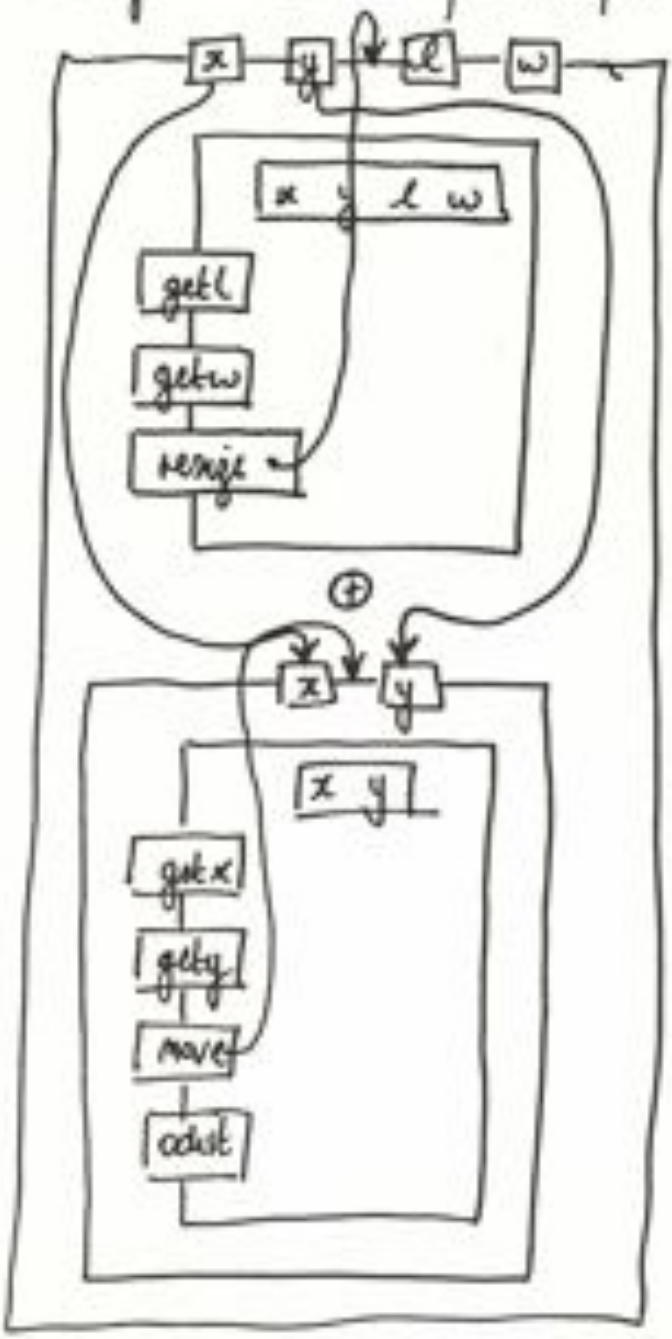
```
let rec make-box (x,y,l,w) =
        { getl = λ().l;
          getw = λ().w;
          resize = λδl. λδw. make-box (x,y,l+δl,w+δw) }
        ⊕ make-point (x,y)
```

But there is a problem,

```
(( make-box (10,20,2,2)).move (1,1)).getl ()
```

produces an error!

To see why look at the definition of the box generator,



Even though the box generator has used the point generator, the move point method returns a new point object when invoked from a box object.

Resize is ok since it refers to the correct generator.

If the point generator is to be used in the box generator then the reference to make-point in move must be changed to make-box.

A Box Generator

This is a problem every time a generator is used in defining a new one. If the box generator is used in defining a cube generator, say, then the make-point generator in more must be changed to make-cube, even though the point generator is not known about in a cube.

Consider making the generator a parameter,

```
let pgen = λgen. λ(x,y).
                 { getx = λ().x ;
                   gety = λ().y ;
                   move = λδx. λδy. gen(x+δx, y+δy) ;
                   odist = λ().√(x²+y²) }
```

Now the point generator can be created by taking the fixed point of pgen.

```
let make-point = fix pgen
```

which returns a function identical to the definition using **rec**.

But now define,

```
let bgen = λgen. λ(x,y,l,w).
                 { getl = λ().l ;
                   getw = λ().w ;
                   resize = λδl. λδw. gen(x,y,l+δl,w+δw) }
                 ⊕ pgen gen (x,y)
```

And the box generator is the fixed point of bgen,

```
let make-box = fix bgen
```

Now the generator can be changed easily in rewriting pgen since the correct function is passed to it as a parameter.

But there is still a problem!

The move method of pgen calls

gen (x+δx, y+δy)

which is ok when the generator is make-point, but when the generator is make-box this expects a 4-tuple, it is supplied with a 2-tuple.

In general the size of tuple cannot be known since any number of extensions can be made, defining more and more horizontally extended generators.

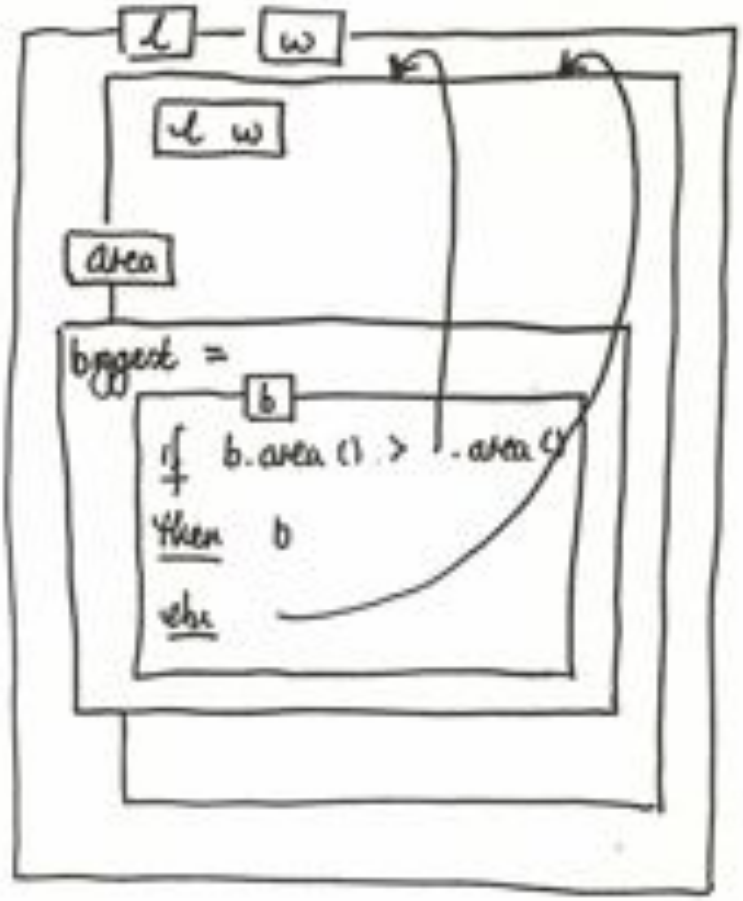
To solve this problem local storage will be "reified", but this cannot be done without introducing self-reference.

14 Self-Reference

Consider a box

```
let make-box = λ(l, w).
                 { area = λ().  l * w ;
                   biggest = λb. if b.area > ME.area()
                                 then b
                                 else ME }
```

Want to refer to the object inside itself.



A box generator.

This could be achieved,

```
let make-box = λ(l, w).
                 let rec self =
                        { area = λ().  l * w ;
                          biggest = λb. if b.area()>self.area()
                                        then b
                                        else self }
                 in
                      self
```

Which would work for boxes. But now extend a box into a bordered box,

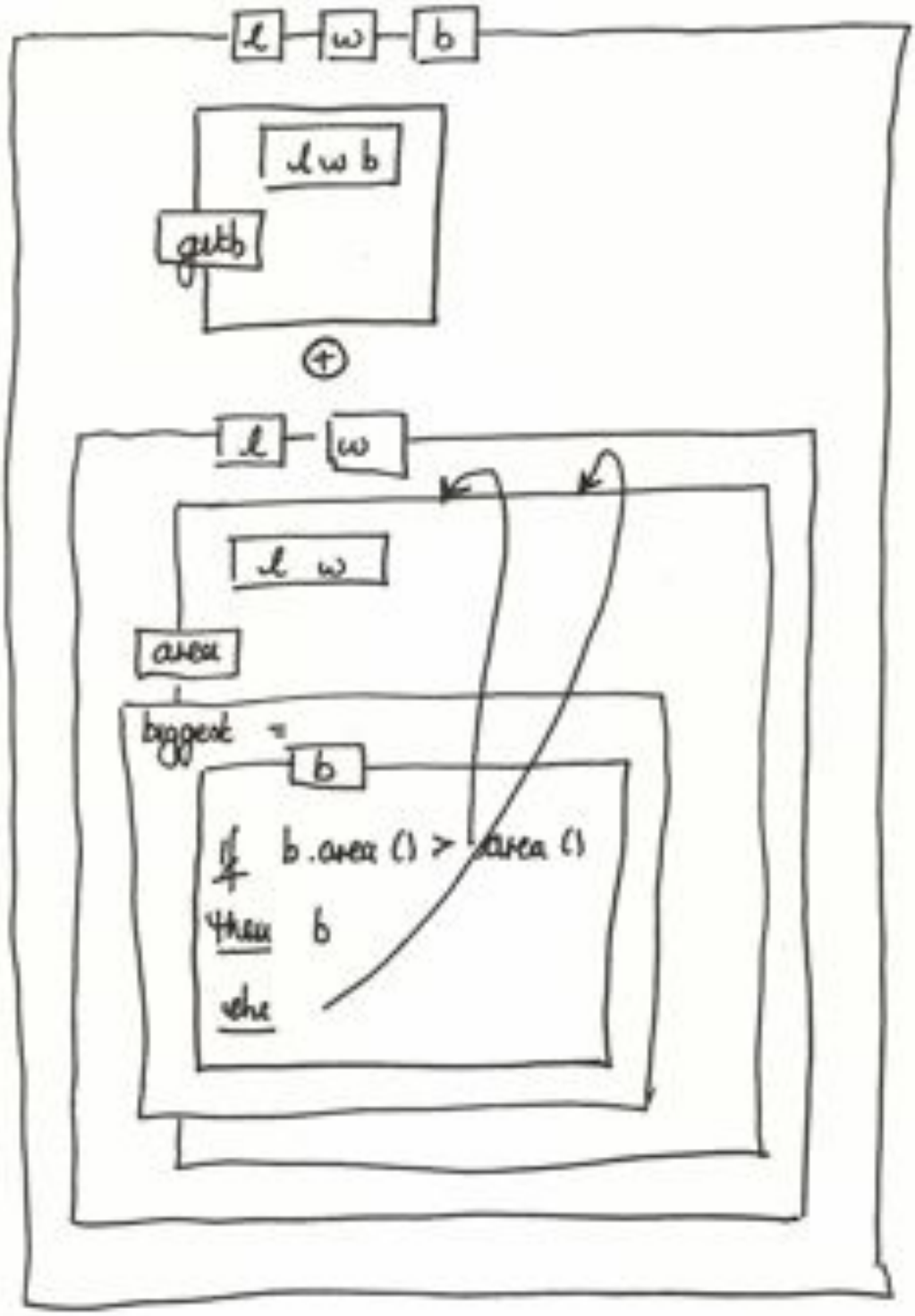
```
let make-bordered-box =
        λ(l, w, b).
            { getb = λ().b }
            ⊕ make-box (l, w)
```

This does not work, consider

```
((make-bordered-box (10,20,1)).biggest
        (make-bordered-box (5,10,1))).getb()
```

ERROR!

A diagram shows what has happened,



A Bordered Box Generator

The self identifier in the box part of a bordered box still refers to the box part.

So, like gen, self cannot be fixed when a generator is defined. It must change each time a generator is extended.

The solution is to make self a parameter which is passed to a generator when an object is created.

15. Self Reference As A Fixed Point

Define the generator to produce a function which takes self as a parameter and return the object which the parameter value represents.

```
let make-box = λ(l,w). λself.
                 { getl = λ().l;
                   getw = λ().w;
                   area = λ().l*w;
                   biggest = λb. if b.area() > self.area()
                                 then b
                                 else self }
```

Now the object is the fixed point of the function returned by the generator

```
let b1 = fix (make-box (10,20))        ①
```

Now define a bordered box in terms of the box generator,

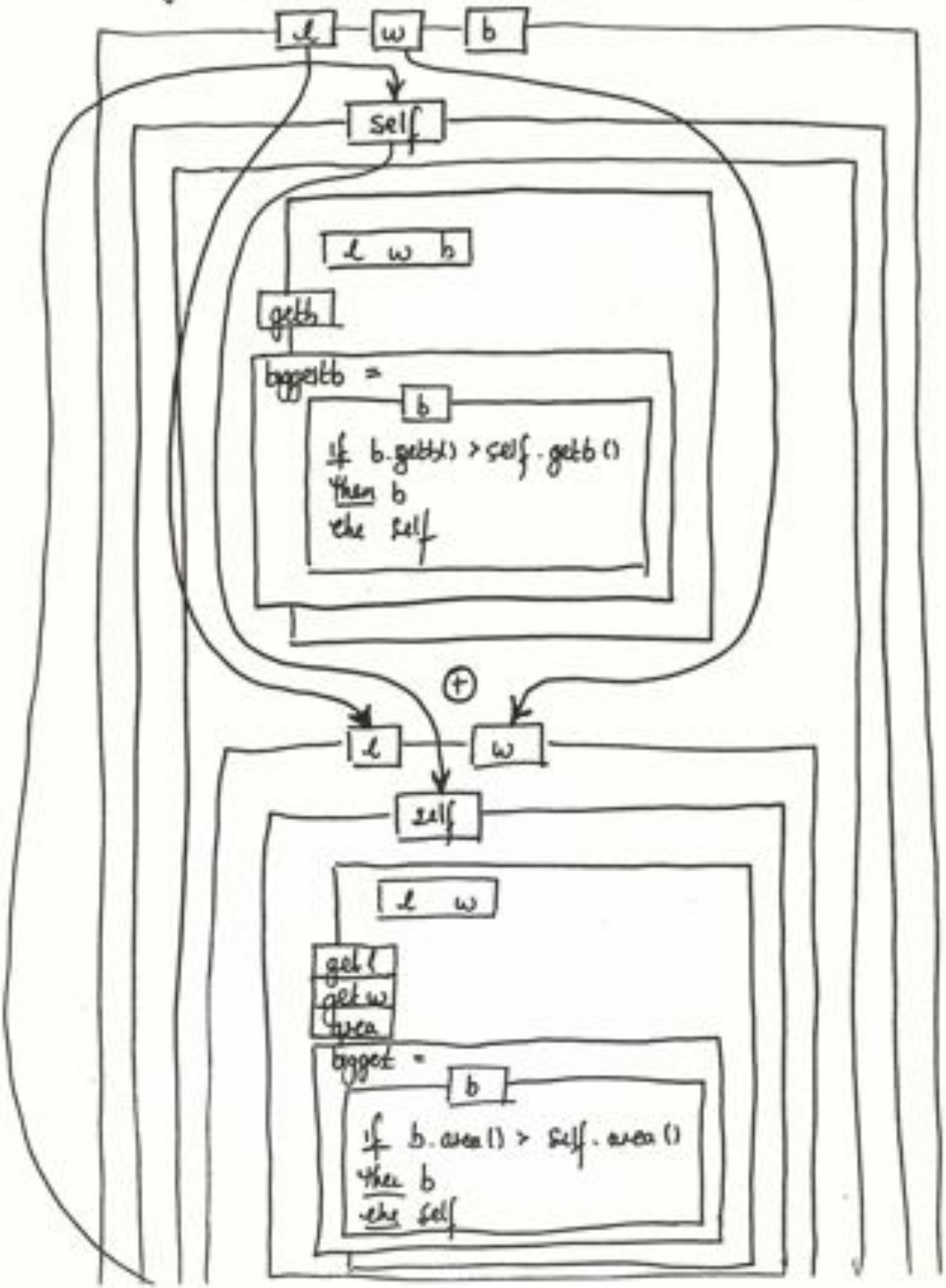
```
let make-bordered-box =
        λ(l,w,b). λself.
            { getb = λ().b;
              biggestb = λb. if b.getb() > self.getb()
                             then b
                             else self }
            ⊕ make-box (l,w) self        ②
```

So the value which is returned by make-box at ① will also be the value of self.

But the value which is returned by make-box at ② will be the value of self as defined by the bordered box — as required.

let bb1 = fix (make-bordered-box (10, 20, 2))

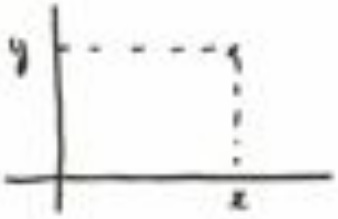
A diagram shows what has happened.



16. Putting Self Reference & Generators Together

A simple example,

Suppose that a generator for a point in 2D space is required,



Given two 2D points can talk about highest & rightmost.

```
let pgen = λgen. λ(x,y). λself.
        { getx = λ().x;
          gety = λ().y;
          highest = λp. if p.gety () > self.gety ()
                        then p
                        else self
          rightmost = λp. if p.getx () > self.getx ()
                          then p
                          else self
          move = λδx. λδy. fix gen (x+δx, y+δy) }

let make-point = fix pgen

let p1 = fix (make-point (10,20))
```

Now use the point generator to define a box (bottom left (0,0)).

Requirements

highest => tallest
rightmost => widest
move => resize
area
biggest.

```
let bgen = λgen. λ(x,y). λself.
            { area = λ().  x * y ;
              biggest = λb. if b.area() > self.area()
                            then b
                            else self } ⊕
            let point = pgen gen (x,y) self and
                highest = point.highest and
                rightmost = point.rightmost and
                move = point.move
            in
                { tallest = highest ;
                  widest = rightmost ;
                  merge = move } ⊕ point
let make-box = fix bgen
let b1 = fix (make-box (10,20))
```

So `(b1.resize (100,200)).area()` will return 2000 and not cause an error.

So `(b1.tallest (fix (make-box (10,19))))` will return b1 as a box <u>not</u> as a point.

17. Local Data Solves The Generator Problem

Recall that as soon as a horizontal extension occurs the calls on the generator from more impoverished generators will cause a problem because they do not supply enough information.

The local data is 'reified' by explicitly storing it within an object and passing it to the generator.

Define 2 new functions to manipulate tuples,

<u>nth</u> t i returns i^{th} element of tuple

<u>with</u> t i v returns a copy of t with the ith element updated to be v.

Define ⊕ extension,

Suppose $\pi 1.x = 1\phi$
$\pi 2.x = 2\phi$

then $(\pi 1 \oplus \pi 2).x = 1\phi$

$(\pi 2 \oplus \pi 1).x = 2\phi$

```
let pgen = λgen. λ(x,y). λself.
        { data = (x,y) ;
          getx = λ()x ;
          gety = λ().y ;
          move = λδx.λδy.
                 let t = with self.data φ (x+δx)
                     t' = with t 1 (y+δy)
                 in
                     fix (gen t')

let make-point = fix pgen

let p1 = fix (make-point (1φ,2φ))
```

```
let bgen = λgen. λ(x, y, l, w). λself.
            { data = (x, y, l, w);
              getl = λ().l;
              getw = λ().w;
              area = λ().l * w }

            ⊕ pgen gen (x,y) self

let make-box = fix bgen

let b1 = fix (make-box (10, 20, 2, 4))
```

13. Redefining Methods

A rectangle has a length & width and is defined in term of a point.

perimeter = 2 * length * width

A Square has a length (& width) and is defined in terms of a rectangle. The perimeter of a square can be computed more efficiently,

perimeter = 4 * length

```
let grec = λgen. λ(x, y, l, w). λself.
            { data = (x, y, l, w);
              getl = λ().l;
              getw = λ().w;
              perm = λ(). 2*l*w } ⊕

            gpoint gen (x,y) self
```

Now define the Square as,

```
let gsquare = λgen. λ(x, y, l). λself.
            { data = (x, y, l, l);
              perim = λ(). 4 * l }

            ⊕ grec gen (x, y, l, l) self
```

19 Late Vs Early Binding

Notice that all methods invoked from <u>within</u> an object indirect through self. This is important to allow subsequent redefinition of the method to take effect.

Consider a bank account,

```
let gbank = λgen. λf. λself.
            { data = f;
              legal-withdrawal = λw. if w > f
                                     then false
                                     else true ;
              withdraw = λw. if legal-withdrawal w
                             then fix gen with self data ↓ (f-w)
                             else error

              query = λ().f ;
              deposit = λd. fix gen with self data ↓ (f+d)
            }
```

An overdraft bank account has an overdraft limit,

```
let gobank = λgen. λ(f,o). λself.
             { data = (f,o);
               legal-withdrawal = λw. if w > f+o
                                      then false
                                      else true } ⊕

gbank gen f self
```

But this does not function correctly.

To see this look at the use of legal-withdrawal in withdraw. The place where it is bound is fixed at the previous field of the record.

When legal-withdrawal is redefined in gobank, this will make no difference to the use in withdraw.

This is called <u>early binding</u>.

To effect <u>late binding</u> change the definition of withdraw to

```
withdraw = λw. if self.legal-withdrawal w
               then fix gen with self.data ∘ (f-w)
               else error
```

20. Private Methods

An object may wish to have methods which it can use but clients cannot.

An example could be legal-withdrawal in the bank account generator.

A meter is an object which measures changes in values. A meter also has a max and min reading.

```
let gmeter = λgen. λ(r, max, min). λself.
        let reg-max = λnew. if new > max
                            then fix gen with (self.data 1 new)
                            else self
        reg-min = λnew. if new < min
                            then fix gen with (self.data 2 new)

        in
            { data = (r, max, min);
              getr = λ().r;
              getmax = λ().max;
```

```
getmin = λ().min;
raise = λδ. let new = fix gen with self.data ← (r+δ)
            in   ??
```

But once a new object has been created its private methods are encapsulated and cannot be invoked.

Solution:

An object has a public and a private part

```
priv o = fst o
pub o = snd o

let gmake = λgen. λ(r, max, min). λself.
        let private =
            { reg-max =
                λr. if r > max
                    then fix gen with (priv self).data 1 r
                    else self ;
              reg-min =
                λr. if r < min
                    then fix gen with (priv self).data 2 r
                    else self ;
              data = (r, max, min) }   and
        public =
            { getr = λ().r ;
              getmax = λ().max ;
              getmin = λ().min ;
```

```
raise = λδ. let new = fix gen with (priv self).data φ (δ+r)
            in pub ((priv new).reg-max (δ+r))
lower = λδ. let new = fix gen with (priv self).data φ (r-δ).
            in pub ((priv new).reg-min (r-δ)) }

in (private, public)

let make-meter = λr. (fix generate) (r, r, r)

let m1 = pub (fix (make-meter φ))
```

A better meter allows the operator to reset the reading.

```
let gbetter meter = λgen. λ(r, max, min). λself.
        let meter = generate gen (r, max, min) self and
            private = priv meter and
            public = { reset =
                       λr. let new = fix gen with (pr
                                   data φ r
                           new' = (priv new).r
                                   1
                           new'' = (priv new').r
                                   r
                           in pub new'' } ⊕
                       pub meter

in (private, public)
```

21 Extending Methods

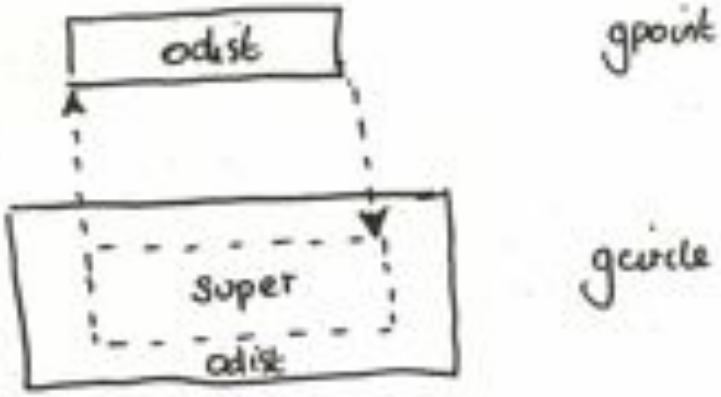
```
let gpoint = λgen. λ(x,y). λself.
        let private = { data = (x,y) } and
            public =
                { getx = λ().x ;
                  getY = λ().y ;
                  move = λδx.λδy.
                      let d = with (priv self.data) δx+δy
                          d' = with d | y+δy
                      in
                      fix gen d' } ;
        in (private, public) odist = λ().√(x²+y²) }
let gcircle = λgen. λ(x,y,r). λself.
        let point = gpoint gen (x,y) self and
            private = {data = (x,y,r)} ⊕ priv point and
            public = { getr = λ().r ;
                       odist = λ().(√(x²+y²)) - r }
                     ⊕ pub point
        in
            (private, public)
```

But circle odist is very like point odist, part of it is identical.

(i) Errors caused by duplication of code

(ii) Code is unnecessarily verbose

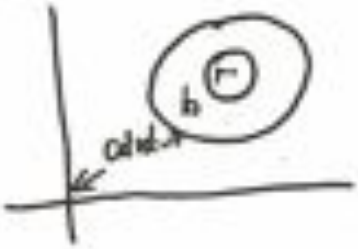
(iii) Performing the gpoint definition of odist as part of gcircle would work if it could



12. Run Super

```
let gcircle = λgen. λ(x,y,r). λself.
                let point = gpoint gen (x,y) self and
                    private = { data = (x,y,r) } ⊕ pri point
                    public = { getr = λ().r ;
                               odist = let super = (pub point).o
                                       in λ(). super() − r }
                                 ⊕ pub point
                in (private, public)
```

Extend further



```
let gbcircle = λgen. λ(x,y,r,b). λself.
        let circle = gcircle gen (x,y,r) self and
            private = { data = (x,y,r,b) } ⊕ priv circle
            public = { getb = λ().b ;
                       clist = let super = (pub circle).
                                           clist
                               in
                               λ(). super() - b }
                     ⊕ pub circle
        in
            (private, public)
```

23. Stripping

```
let gnetc = λstrip. λgen. λ(r, max, min). λself.
        let private =
            { data = (r, max, min) ;
              neg-max = λr. if r > max
                            then fix gen unth (priv self) data
                                 | r
                            else self
              neg-min = λr. if r < min
                            then fix gen unth (priv self). data
                                 2 r }
        public =
            { getr = λ().r ;
              getmax = λ().max ;
              getmin = λ().min ;
              rotate = λδ. let new = fix gen unth
                                 (priv self). data
                                 ⊕ r+δ
                       in
                       strip (priv new). neg-max r+δ
```

```
lower = λδ. let new = fix gen unit (priv self).data ∘ r-δ
            in
                skip (priv new).reg-min r-δ}

in
    (private, public)
let make-mate = λr. (fix (gmate pub)) (r,r,r)

let gmate = λskip. λgen. λ(r, max, min, c). λself.
        let smate = gmate id gen (r, max, min) self and
            mate = gmate pub gen (r, max, min) self and
            private =
                { data = (r, max, min, c) ;
                  inc = λ0. fix gen unit (priv self).data 3 c+1
                ⊕ priv meter
            public =
                { get c = λ(). c ;
                  raise = let super = (pub smate).raise
                          in λδ. let new = super δ
                                 in skip ((priv new).inc (),
                  lower = let super = (pub smate).lower
                          in λδ. let new = super δ
                                 in skip ((priv new).inc (
                ⊕ pub meter
        in (private, public)
```

(6b)

<u>let</u> make-counter-meter =

$\lambda r.$ (<u>fix</u> (gometer <u>pub</u>)) (r, r, r, ϕ).

PDF