CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Existential Types II

Larger ADT Example

```
type env ≡ ∃α.{empty : α,
               lookup : α -> id -> val,
               extend : α -> id -> val -> α}

list_env =
  pack [(id * val) list,
       {empty = [],
        lookup = λe:(id*val) list. λ x:id. list_assoc e x,
        extend = λe:(id*val) list. λ x:id. λ v:val.(x,v)::e
       }]
  as env

func_env =
  pack [id -> val,
       {empty = λy:id. raise Empty,
        lookup = λe:(id->val). λ x:id. e x,
        extend = λe:(id->val). λ x:id. λ v:val.
                        λy:id. if x = y then v else e y
       }]
  as env

client = λenv_impl:env.
           open env_impl as [α,ei] in
             ei.lookup "p" (ei.extend ei.empty "q" 10)

prog1 = client list_env

prog2 = client func_env
```


Existentials and Subtyping

We can add a simple rule for subtyping between existential packages:

$$\frac{\tau' <: \tau}{\exists\alpha.\tau' <: \exists\alpha.\tau}$$

This gives one basis for solving the problem with OO encodings we saw before.

Suppose we try to encode these class definitions into an explicit representation in terms of records and functions.

```
class Point {
  int x,y;
  draw() { output(x,y,BlackPen); }
}

class ColorPoint {
  int x,y;
  color c;
  draw() { output(x,y,c); }
}
```

As before, we give each method an extra `self` parameter; but, to avoid issues with recursive types, we just pass the **fields** of each object as its self value.

Failed OO Encoding

This gives the following encodings:

```
Point p = new Point(x0,y0) ⟶
p : {fields:{x:int, y:int},
     draw: {x:int,y:int} → unit}
  = {fields = {x = x0, y = y0},
     draw = λself:{x:int, y:int}.output(x,y,BlackPen)}

ColorPoint cp = new ColorPoint(x0,y0,c0) ⟶
cp : {fields = {x:int, y:int, c:color},
      draw: {x:int,y:int,c:color} → unit}
   = {fields = {x = x0, y = y0, c = c0}
      draw = λself:{x:int, y:int, c:color}.output(x,y,c)}

p.draw() ⟶
p.draw(p.fields)

cp.draw() ⟶
cp.draw(cp.fields)
```

Recall problem:

- We want the type of `cp` to be a subtype of the type of `p`, so that we can use a `cp` wherever a `p` is wanted.
- But the `self` argument to `draw` is covariant, so the desired subtype relationship doesn't hold. In particular, it would be a big mistake to try

```
cp.draw(p.fields)
```

- But of course, we don't really plan to make calls like that!

Using Existentials

One approach to solving this problem is to treat the `self` record as an **abstract** type, and package the object as an existential.

```
Point p = new Point(x0,y0) ⟶
p =
  pack [{x:int,y:int},
       {fields = {x = x0, y = y0},
        draw = λself:{x:int, y:int}.output(x,y,BlackPen)}]
  as ∃α.{fields:α, draw : α → unit }

ColorPoint cp = new ColorPoint(x0,y0,c0) ⟶
cp =
  pack [{x:int,y:int,c:color} ,
       {fields = {x = x0, y = y0, c = c0},
        draw = λself:{x:int, y:int, c:color}.output(x,y,c)}]
  as ∃α.{fields:α, draw : α → unit }

p.draw() ⟶
open p as [β,p'] in
  p'.draw(p'.fields)

cp.draw() ⟶
open cp as [β,cp'] in
  cp'.draw(cp'.fields)
```

Now we have that the type of `cp` is a subtype of the type of `p`, as desired. And the dangerous problem with covariance on `self` cannot occur, because the *only* thing we can do with a record representing `self` is to pass it a method of its own class!