Family polymorphism

December 7, 2019

Family

The receptionist decides to get things going by asking a man "Are you a husband?" and asking a woman "Are you a wife?". Upon receiving two affirmative - though slightly baffled - answers, those two people are assigned to the same room, together with a little girl who said "Erm, yeah, and I'm a daughter!"

Goal

- Group of classes forming a family
- Unbounded amount of those families
- Type safety: Ability not to mix up families
- Flexibility of using any of those family

Examples

- Graph
- Subscriber/Observer
- Family room assignment

Family room assignment

```
class Husband { };

class Wife { };

class Room {
  def assign_husband(h: Husband) : Room = this;
  def assign_wife(w: Wife) : Room = this;
};

val husband = new Husband();
val wife = new Wife();

val room42 = new Room();

room42.assign_husband(husband);
room42.assign_wife(wife);
```

```
class Father extends Husband { };
class Mother extends Wife { };

val husband = new Husband();
val wife = new Wife();
val alsohusband = new Father();
val alsowife = new Mother();

val room42 = new Room();

room42.assign_husband(husband);
room42.assign_wife(wife);

room42.assign_husband(alsohusband);
room42.assign_wife(wife);
room42.assign_husband(husband);
room42.assign_wife(alsowife);
```

```
template<typename W, typename H>
struct AbstractWife { int num_child() { return 0; } };

template<typename W, typename H>
struct AbstractHusband { };

struct Wife;
struct Husband : public AbstractHusband<Husband, Wife> {};
struct Wife : public AbstractWife<Husband, Wife> { };

struct Mother;
struct Father : public AbstractHusband<Father, Mother> { };
struct Mother : public AbstractWife<Father, Mother> {
  int number_of_child;
  int num_child() { return number_of_child; }
};
```

```
template <typename H, typename W> struct Room {
  W *wife;
  void assign(H *h, W *m) { wife = m; }
  int room_size() { return  wife->num_child()+2; };
};

int main() {
  auto *room42 = new Room<Husband, Wife>();
  auto *room43 = new Room<Father, Mother>();

  room42->assign(new Husband(), new Wife());
  //room42->assign(new Husband(), new Mother());
  room42->room_size();

  room43->assign(new Father(), new Mother());
  //room43->assign(new Husband(), new Mother());
  room43->room_size();
};
```

Type variance

```
C { type t = T }  // if t is declared non-variant,
C { type t <: T } // if t is declared co-variant,
C { type t >: T } // if t is declared contra-variant.
```

Parameter variance

```
class C[T]  { } // if t is declared non-variant,
class C[+T] { } // if t is declared co-variant,
class C[-T] { } // if t is declared contra-variant.
```

```
class AbstractFamily {
  type Husband <: AbstractHusband
  type Wife    <: AbstractWife

  abstract class AbstractHusband { };
  abstract class AbstractWife { };
};

class Family extends AbstractFamily{
   type Husband = YoungHusband
   type Wife    = YoungWife

   class YoungHusband extends AbstractHusband { };
   class YoungWife extends AbstractWife { };
};

class FamilyWithKids extends AbstractFamily {
  type Husband = Father
  type Wife    = Mother

  class Father extends AbstractHusband {
     def has_child: Boolean = true;
  }

  class Mother extends AbstractWife {
     def has_child: Boolean = true;
  }
};

abstract class Room {
  type F <: AbstractFamily

  def assign_husband(h: F#Husband): Room = this;
  def assign_wife(w: F#Wife): Room = this;
};
```

```
val family = new Family();
val husband = new family.Husband();
val wife = new family.Wife();

val room42 = new Room { type F = Family; };
val room43 = new Room { type F = FamilyWithKids; };

room42.assign_husband(husband);
room42.assign_wife(wife);

room42.assign_husband(husband2);

val marriedfamily = new FamilyWithKids();
val alsohusband = new marriedfamily.Husband();
val alsowife = new marriedfamily.Wife();

assert(alsohusband.has_child);
// This would not compile as the husbend
// is just Husband end not a Father
//assert(husband.has_child);

room43.assign_husband(alsohusband);
room43.assign_wife(alsowife);

//Does not compile
room43.assign_wife(wife);

room42.assign_husband(alsohusband);
room42.assign_wife(wife);

room42.assign_husband(husband);
room42.assign_wife(alsowife);
```

Graph

- Scala
- gBETA

```
abstract class Graph {
  type Node <: AbstractNode
  type Edge <: AbstractEdge

  def mkNode() : Node
  def connect(n1: Node, n2: Node) : Edge

  abstract class AbstractEdge(val n1: Node, val n2: Node)

  trait AbstractNode {
    def touches(edge: Edge): Boolean = {
        edge.n1 == this || edge.n2 == this
    }
  }
}
```

```
class BasicGraph extends Graph {
  type Node = BasicNode
  type Edge = BasicEdge
  protected class BasicNode extends AbstractNode
  protected class BasicEdge(n1:Node, n2:Node)
            extends AbstractEdge(n1, n2)

  def mkNode() = new BasicNode
  def connect(n1: Node, n2: Node) : BasicEdge = {
    new BasicEdge(n1, n2)
  }
}


class OnOffGraph extends Graph {
  type Node = OnOffNode
  type Edge = OnOffEdge
  protected class OnOffNode extends AbstractNode {
    override def touches(edge: Edge): Boolean = {
      edge.enabled && super.touches(edge)
    }
  }
  protected class OnOffEdge(n1:Node, n2:Node,
                            var enabled: Boolean)
            extends AbstractEdge(n1, n2)

  def mkNode() = new OnOffNode
  def connect(n1: Node, n2: Node) : OnOffEdge = {
    new OnOffEdge(n1, n2, true)
  }
}
```

```
val g = new BasicGraph
val n1 = g.mkNode()
val n2 = g.mkNode()
val e = g.connect(n1, n2)
assert(n1 touches e)
assert(n2 touches e)
val g2 = new BasicGraph
//g2.connect(n1, n2) // Does not compile

val og = new OnOffGraph
val on1 = og.mkNode()
val on2 = og.mkNode()
val oe = og.connect(on1, on2)
// val mixed = og.connect(n1, n2) // ERROR: og.connect not applicable to g.Node

assert(on1 touches oe)
assert(on2 touches oe)
// println(on2 touches e) // ERROR: on2.touches not applicable to g.Edge
oe.enabled = false;
assert (! (on2 touches oe), "After disabling, edge virtually has gone")
assert (! (on1 touches oe), "After disabling, edge virtually has gone")
```

```
def addSome(graph: Graph): Graph#Edge = {
  val n1, n2 = graph.mkNode()
  graph.connect(n1, n2)
}
val g = new BasicGraph
val og = new OnOffGraph

val e2 = addSome(g)
val oe2 = addSome(og)
// oe2.enabled = false // type OnOffGraph not retained, graph.Edge not possible
```

```
def addSome2[G <: Graph](graph: G): graph.Edge = {
  val n1, n2 = graph.mkNode()
  graph.connect(n1, n2)
}

val g = new BasicGraph
val og = new OnOffGraph

val e22 = addSome2(g)
val oe22 = addSome2(og)
oe22.enabled = false // now OK.
```

```
(# Graph:
   (# Node:<
      (# touches:<
         (# e: ^ Edge; b: @boolean
            enter e[]
            do (this(Node)=e.n1) or (this(Node)=e.n2)->b
            exit b
         #);
         exit this(Node)[]
      #);
      Edge:< (# n1,n2: ^ Node exit this(Edge)[] #)
   #);
   OnOffGraph: Graph
   (# Node::< (# touches::<!(# do (if e.enabled then INNER
                                    if)#)#);
      Edge::< (# enabled: @boolean #)
   #);

   build:
   (# g:< @Graph; n: ^ g.Node; e: ^ g.Edge; b: @boolean
      enter (n[],e[],b)
      do n->e.n1[]->e.n2[];
      (if (e->n.touches)=b then 'OK'->putline if)
   #);

   g1: @Graph; g2: @OnOffGraph
   do
     (g1.Node, g1.Edge, true) -> build(# g::@g1 #);
     (g2.Node, g2.Edge, false) -> build(# g::@g2 #);
     (* type error *)
     (* (g2.Node, g1.Edge, false) -> build(# g::@g1 #); *)
     (* (g2.Node, g1.Edge, false) -> build(# g::@g2 #); *)
#)
```