Explore Family polymorphism

Péter Kókai

Eötövs Loránd University

Keywords polymorphism, family polymorphism

1 Introduction

There are different type of polymorphism present, while there are a well known and commonly used type of polymorphism such as parametric, subtype or ad-hoc polymorphism, the research did not stop there. The aim of this paper is not to create a yet another polymorphism rather do a deep dive on the family polymorphism. This type of polymorphism was first mentioned by Erik Ernest expanding the BETA language into gBETA. The concept was later adapted by Scala as well.

The examples mostly going to be using Scala programming language. This paper assumes the reader are familiar with Scala at basic level, and a few concepts will be explained only when required. For detailed overview of Scala check the following works[12][9].

The next section is going to be an introduction to the currently dominating type of polymorphism, and basic building blocks that are used to create a family polymorphism including virtual classes, self-types.

The 3rd section is finally introduces the main topic of this paper family polymorphism itself via problems that were partially already introduced in the first section. Providing a possibly sound solution.

After that exploring the lightweight family polymorphism compared to family polymorphism.

In the end one must speak and reason about the type system used by languages supporting family polymorphism such as gBETA or Scala.

2 Polymorphism

Among popular and relatively old languages such as Java, C++ or C# it is possible to see a polymorphism is a key feature. The most common object-oriented languages provide subtyping, that is a well known type of polymorphism sometimes also referred as inclusion polymorphism. As its name suggest - inclusion - has other benefits of this mechanism is to share code, specifically avoid code duplication. Of course code duplication sometimes acceptable, still it is in the coders best interest to avoid it. Despite the subtyping popularity, that is only a type of polymorphism. Finding different types in the same language also not uncommon. Just as provided example for subtype polymorphism both Java and C++ has a means to provide a different way to achieve multiple forms. Just think of template metaprogramming or generic introduced in Java. Those provides a means to write generic function, code via a parametric polymorphism.

2.1 Ad-hoc polymorphism

It is possibly to provide a function or method with one name, but in reality refer to multiple implementation depending on the different type or number of arguments. This is also referred to in languages as function overloading or method overloading. Such an example is:

```cpp
#include <cmath>

bool even(const int n) {
  return 0 == n % 2;
}

bool even(const double n) {
  return 0 == std::mod(n,2);
}
```

In this case the *even* has two different implementation for different types, the dispatch is made in compile time therefore only the static types are checked. By using the same name has its own risk of producing unreadable code, but exploring readability issues is not a scope now.

2.2 Parametric polymorphism

In contrast with ad-hoc where the type could have completely different, the parametric polymorphism aim is to create one implementation that works with different types while still keep the type safety, as in the following example shows the static type is used to determine if the dispatching is possible.

```cpp
#include <cmath>

template<typename Number>
bool even(Number n) {
   return 0 == std::fmod(n, 2);
};
```

In this example only one *even* function can be seen, while the trick here is to use std::fmod that itself has an ad-hoc definition in C++. It is possible to provide an implementation without std::fmod, this was kept for because of its length.

The *Number* itself cannot be any type, as there is still a restriction as it requires the std::fmod to be implemented for that type. Those type of restriction could also be specified in c++ via concepts[2]. The parametric polymorphism also known as in many languages as Generic programming, that approach was taken in Java or Scala.

2.3 Subtype polymorphism

The subtype describes a relation between two types, in fact if a S type is a subtype of T (in short S<:T), that means that using S instead of T should be transparent and safe to do[7]. This is usually referred as only polymorphism in OO languages, and used with combination of inheritance providing flexible design and help code reuse.

Following the example before with implementing an *even* function for numbers.

```
class Number { };

class Integer : public Number { };
class Double : public Number { };


bool even(Number n) {
  return Number(0) == n.fmod(2);
}
```

Without providing the implementation details let's assume it is so Integer<:Number and Double<:Number are both true. The implementation of *even* works for all S type that is subtype of the Number, and inherited from either Number or a type inherited from Number and still a subtype of Number. It is easy to confuse the inheritance with subtype in this example.

3 Additional techniques

3.1 Virtual classes

The usage of virtual functions and method in object oriented languages are common, and well understood.

In language like C++, the abstract class can be defined via creating a virtual function not providing implementation for those functions, thus forming a kind of interface for subclasses. Just like in the example with *Number* the *fmod* function must be either using virtual function or be one itself. Because the *even* function has no knowledge about the exact type or name of that type that the field going to have in either its *Integer* or *Double*.

While from *Number* point of view it would be possible not to even hold any value in its subclasses. But the signature of the *fmod* clearly states otherwise. The reader of such class not holding value could be surprised, as it would most likely seem unnatural.

Of course solving such issue is possible with the combination of parametric polymorphism:

```
template<typename Storage>
class Number {
  Storage value;
};

class Integer : public Number<int> {};
class Double  : public Number<double> {};
```

Even with extending the *Number* with the *value* without knowing all of its subclasses that finally specifies the *Storage* it is impossible to do a full analysis on that class. The possibilities with the current sets of C++ stops at this point.

Despite that C++ failed us in this example, by checking out the current sets of features. Specifically the virtual method calls. The dynamic dispatching of the operation defined for objects only deals with the functions and methods, while the object itself contains also fields holding values, and defining their own set of operations. Those operation could in fact define their own set of operation, but the type of the fields are still limited to from the first parent in the inheritance chain. There is no way with using standard inheritance to further refine the type of the fields of the objects. In order to solve template related issues one should check out concepts[2]. But that is not the topic that interested here.

The simple idea is to extend objects with field that could hold a class as a value. The key difference between this and nested classes that nested classes are classes that tied to the inner class and those are not possible to modify dynamically, no late binding is possible. By allowing the class to be tied to the object itself makes it possible to defer type information into run-time. Additionally the objects are safe to store not only the class but also a subclass. This dynamic dispatching can be called as virtual classes.

4 Family polymorphism

The family polymorphism described here aims to scale the current polymorphism into the multi object, method level. Where multiple object and methods can be combined in a type safe manner.

It is easy to mix up families just like the example with three person describes from the introduction of family polymorphism[3]:

> 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!"

Even if those people has those roles in a family, it does not necessarily means that they are part of the same one. Thus assigning them for the same room might not be the best idea. With the above idea let first just focus on the husband and wife as it is going to be an issue in case of two object, by allowing an additional third object only complicates the example and does not provide any solution. Just lets craft a husband, a wife and a room in Scala as follows:

```scala
class Husband { };

class Wife { };

class Room {
  var husband: Husband = null;
  var wife: Wife = null;

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

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

val room42 = new Room();

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

There is nothing uncommon in that example, it does not even introduces the possibility of other husbands and wives. Still the wife and the husband can be considered to be part of the same family. Everything works as expected the family is in the *room42* and happy.

Mixing up things with only one room and one family is really hard. By introducing multiple rooms the situation still controllable as being in a different room might not be ideal but okay.

The issue raises when a second family arrives, and now it is a father and a mother. This other family also consists of Husband and Wife, but they are clearly part of a different family as they have a child while the other family before them did not.

```
class Father extends Husband {
  def has_child(): Boolean = true;
};

class Mother extends Wife {
  def has_child(): Boolean = true;
};

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

val room42 = new Room();
val room43 = new Room();

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

val alsohusband = new Father();
val alsowife = new Mother();

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

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

room43.assign_husband(husband);
room43.assign_wife(alsowife);
```

Just like anybody would expected from inheritance it is perfectly fine to assign the father and the wife into the same room, while that is clear that they are from a different family.

It was shown in the original paper that it is possible to solve this kind of issues with the existing polymorphism, but there is no solution that both kept type safety and reusability. Of course such a relationship between the families can be severed with not inheriting from the same base class, rendering the wife and the husband and the mother and the father irrelevant classes. Any object-oriented type system would detect those classes unrelated. Such a solution would solve the type safety, but would not allow reusability. The trial of such resolution could be checked in Ernst's paper[3].

The need to connect those objects that are part of the same family is needed. Both Java and C++ supports nested classes, in those languages the nested class itself does not solve the issue. The types of those classes would be the same just wrapped with an other class.

The Scala example with nested classes:

```
class AbstractFamily {
  class AbstractHusband { };
  class AbstractWife { };
};

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

class FamilyWithKids extends AbstractFamily {
  class Father extends AbstractHusband {
    def has_child(): Boolean = true;
  };
  class Mother extends AbstractWife {
    def has_child(): Boolean = true;
  };
};

class Room {
  type AH = AbstractFamily#AbstractHusband;
  type AW = AbstractFamily#AbstractHusband;

  def assign_husband(h: ): Room = this;
  def assign_wife(w: AW): Room = this;
};


val family = new Family();
val marriedfamily = new FamilyWithKids();
val room42 = new Room;

val husband = new family.YoungHusband();
val alsowife = new marriedfamily.Mother();
```

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

This still works as effectively nothing change, only the classes get embedded into a class. But Father <: AbstractHusband still true. The solution would require that both FamilyWithKids <: AbstractFamily, Father <: AbstractHusband and Mother <: AbstractWife are true, and the same with Family <: AbstractFamily, YoungHusband <: AbstractHusband and YoungWife <: AbstractWife while mixing the YoungHusband and Mother or Father and YoungWife would not be possible.

In Scala it is possible to include a type as a field of an object, those fields works effectively just like any other type name. Except they are bound to the object itself, and even two object with the same class having the same type as field when accessed via the objects are observed as different types, as the type system is path dependent. Therefore the objects are also part of the type name, causing different objects having different types.

4.1 Virtual type in scala

This field in scala is defined via the *type* keyword. The type definition could be deferred for later classes simple not specifying the exact type, just like in this example shown:

```scala
abstract class AbsCell {
  type T;
  val init: T;

  private var value: T = init;
  def get: T = value;
  def set(x: T): Unit = { value = x }
}

object cel extends AbsCell {
  type T = Integer
  val init = 14
}

cel.set(cel.init)
println(cel.get);
```

That abstract type later on in the inheritance chain could further refined and at some point of the chain the value (at least before instantiate) must be fixed. The further refinement is possible as non-, co-, and contra-variant:

```scala
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.
```

4.2 Family polymorphism in scala

The *AbstractFamily* could be improved by providing not only AbstractHusband and AbstractWife but an abstract type for those pairs that are co-variant with their Abstract pair.

Check out only the differences in the class definition. The unchanged lines are emitted from the next example.

```
class AbstractFamily {
  type Husband <: AbstractHusband
  type Wife    <: AbstractWife
  ...
};

class Family extends AbstractFamily{
   type Husband = YoungHusband
   type Wife    = YoungWife
   ...
};

class FamilyWithKids extends AbstractFamily {
  type Husband = Father
  type Wife    = Mother
  ...
};

abstract class Room {
  type F <: AbstractFamily

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

With the following change changing the *Room* so it could hold A family instead of members of a family. Please note that while the previous *Room* implementation would still work the goal here is to only work within the same family. The path dependent types in the *Room* now fixed once a family is fixed, and it is not possible to instantiate a non-fixed family.

```
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);

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

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

```
//Mixing families do not work
//Does not compile
room42.assign_husband(alsohusband);
room42.assign_wife(alsowife);
```

The solution presented here has some limitation compared to the original family polymorphism, that difference has impact on both the type system required and language features needed to implement. Before exploring that difference, let me share an other example, which does not suffer of such limitations.

4.3 Original Graph example in gBETA

The introduction of family polymorphism was originally in gBETA[4] language. The gBETA is an improved version of the BETA language. In the BETA language, classes and methods has been merged into a term patterns. The virtual method became virtual pattern, and with that virtual classes became possible. Compared to Scala, Java or even C++ the virtual methods in BETA/gBETA are not overridden by subclasses rather refined by the subclass implementation. The INNER keyword is used to refer to parent implementation.

```
(# 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#);
  (* compile error *)
  (*(g2.Node,g1.Edge,false)->build(#g::@g1#);*)
  (*(g2.Node,g1.Edge,false)->build(#g::@g2#);*)
#)
```

For the eyes that are unfamiliar with the gBETA/BETA syntax the following code describes Graph and OnOffGraph families. They both uses edge representation, and the OnOffGraph - as its name suggest - could switch its edges. Additionally implements a *touches* method displaying *OK* on the console if two *Node* has an edge connecting them.

The same could be written in Scala also. While in Scala it is not that compact as in gBETA, it maybe easier for the reader to understand what happens.

```
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)
//ERROR: og.connect not applicable to g.Node
//val mixed = og.connect(n1, n2)

assert(on1 touches oe)
assert(on2 touches oe)
//ERROR: on2.touches not applicable to g.Edge
// println(on2 touches e)
oe.enabled = false;
assert (! (on2 touches oe))
assert (! (on1 touches oe))
```

The graph Scala implementation was published in the following blog post[6]. That post also describes family polymorphism in details limited to Scala.

Other examples that could illustrate the need for the family polymorphism are only mentioned here. These examples includes the Subscriber/Publisher described in the Scala overview[12] and the extendable compiler[10]. The Subscribed/Publisher example is pattern used in many graphical applications.

5 Lightweight family polymorphism

The solution presented in gBETA yields a complex type system. There are *vObj* or *vc* describe later. Those type system could help proving the soundness of the original family polymorphism. The class of families could be achieved in a more relax way, while the type safety and the reusability is kept as desired. This is usually refereed as lightweight family polymorphism, it does not require for the objects to hold their own distinct type class rather allows classes to purely form families. Still nested classes are not enough to solve lightweight version. Just like the original family polymorphism it is possible to use it in Scala, in fact it is really easy to mix up them in Scala.

An experienced Scala programmer could point out the example with families has such restriction because of how the *Room* is crafted. It only referred to the family with the path *class#type_field*, instead of using the object as type prefix.

Relative path types

```
class Graph {
  static class Node {};
  static class Edge { .Node n1, n2; };
};
```

It is possible to reference to the Node as Graph.Node anywhere and that specify the exact type. Inside the family of Graph, it is also possible to refer the class of a family with a *relative path type* emitting the *Graph* and simply using *.Node* while the *Graph.Node* is called *fully qualified path type*.

Inheritance without subtyping.

```
class OnOffGraph extends Graph {
  static class Node { };
  static class Edge { };
};
```

This means that OnOffGraph.Node inherits all of the Graph.Node's properties still the OnOffGraph.Node is not a subclass of the Graph.Node.

6 Type system soundness

6.1 vOjb

The concepts and type system of advanced languages like BETA, gBETA or Scala are far advanced a simple type calculus. There was a need to create a possible ways to reason about the soundness[13] of those patterns. There were several works that aim to provide such theory introducing embedded types within objects, classes describing nested structures, and using pattern as a general term for class, method, function.

The vObj[11] were created to deal with dependent types. The vObj follows the BETA by defining the Class $\bar{M}$ethod $\bar{F}$unctor, the same ideology of pattern instead of differentiating those.

6.2 vc: virtual class calculus

The OO has concepts for virtual methods or functions, that has the ability to defer the decision which method should be called in run-time, as the method itself is dependent on the object and not the static type of the object(class). This provides a flexibility for the code owner to handle the object and not the class. The objects are not only sets of methods and functions, they also includes attributes. These attributes in most cases are statically typed and fixed at compile time. This restriction can be lifted in order to provide the same citizenship as for the methods, and it is called virtual classes. The virtual classes were implemented as virtual patterns in the BETA[8] language as well as in Ceaser[1]. While BETA provides an implementation for the virtual classes, it was never proven that the type system is sound. There was a need to reason about virtual classes in order to prove its type system is complete, so a virtual class calculus[5] were created to deal with virtual classes. The paper both introduced static and dynamics of vc but also proved its soundness.

References

[1] Ivica Aracic et al. “An overview of CaesarJ”. In: *Transactions on Aspect-Oriented Software Development I*. Springer, 2006, pp. 135–173.

[2] Gabriel Dos Reis and Bjarne Stroustrup. “Specifying C++ concepts”. In: *ACM SIGPLAN Notices*. Vol. 41. 1. ACM. 2006, pp. 295–308.

[3] Erik Ernst. “Family polymorphism”. In: *European Conference on Object-Oriented Programming*. Springer. 2001, pp. 303–326.

[4] Erik Ernst. “gbeta-a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance”. In: *DAIMI Report Series* 549 (2000).

[5] Erik Ernst, Klaus Ostermann, and William R Cook. *A virtual class calculus*. Vol. 41. 1. ACM, 2006.

[6] Martin Kneißl. *Family Polymorphism in Scala*. Avaiable from web.archive.org. 2015. URL: `http://www.familie-kneissl.org/Members/martin/blog/family-polymorphism-in-scala`.

[7] Barbara Liskov. “Data abstraction and hierarchy”. In: *SIGPLAN notices* 23.5 (1988), pp. 17–34.

[8] Ole Lehrmann Madsen and Birger Moller-Pedersen. *Virtual classes: a powerful mechanism in object-oriented programming*. Vol. 24. 10. ACM, 1989.

[9] Nathaniel Nystrom, Stephen Chong, and Andrew C Myers. “Scalable extensibility via nested inheritance”. In: *ACM SIGPLAN Notices*. Vol. 39. 10. ACM. 2004, pp. 99–115.

[10] Martin Odersky and Matthias Zenger. “Scalable component abstractions”. In: *ACM Sigplan Notices*. Vol. 40. 10. ACM. 2005, pp. 41–57.

[11] Martin Odersky et al. “A nominal theory of objects with dependent types”. In: *European Conference on Object-Oriented Programming*. Springer. 2003, pp. 201–224.

[12] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. 2004.

[13] Andrew K Wright and Matthias Felleisen. “A syntactic approach to type soundness”. In: *Information and computation* 115.1 (1994), pp. 38–94.