1 Hierarchy of Classes

```
object pets {
  trait Pet
  trait Dog extends Pet
  trait Cat extends Pet
  trait Poodle extends Dog
  trait Dalmatian extends Dog
}
```

val pets = new ⊤ { z ⇒
  Pet_c: ⊥ .. ⊤,
  Dog_c: ⊥ .. z.Pet_c,
  Cat_c: ⊥ .. z.Pet_c,
  Poodle_c: ⊥ .. z.Dog_c,
  Dalmatian_c: ⊥ .. z.Dog_c }()

1.1 Typechecking Properties

true
e-subtype[[pets.Dog_c, pets.Pet_c]]

true
e-subtype[[pets.Poodle_c, pets.Dog_c]]

2 Type Members

```
object choices {
  trait Alt {
    type C
    type A <: C
    type B <: C
    val choose : A => B => C
  }
}
```

val choices = new ⊤ { z ⇒
  Alt_c: ⊥ .. ⊤ { alt ⇒
    C: ⊥ .. ⊤,
    A: ⊥ .. alt.C,
    B: ⊥ .. alt.C,
    choose: alt.A →arrow[[alt.B, (alt.A ∨ alt.B)]] } }()

2.1 Typechecking Properties

false
e-subtype[[choices.Alt_c { alt ⇒ C: pets.Dog_c .. pets.Dog_c },
  choices.Alt_c { alt ⇒ C: pets.Pet_c .. pets.Pet_c }]]

true
e-subtype[[choices.Alt_c { alt ⇒ C: ⊥ .. pets.Dog_c, A: alt.C .. alt.C, B: alt.C .. alt.C },
  choices.Alt_c { alt ⇒ C: ⊥ .. pets.Pet_c, A: alt.C .. alt.C, B: alt.C .. alt.C }]]

3 Polymorphic Operators

```
object Main extends App {
  import pets._
```

```
  import choices._

  def pickLast[Cp,Ap<:Cp,Bp<:Cp] = new Alt {
    type C = Cp
    type A = Ap
    type B = Bp
    val choose: A => B => B = a => b => b
  }

  val potty = new Poodle { override def toString = "potty" }
  val dotty = new Dalmatian { override def toString = "dotty" }

  val picker = pickLast[Dog,Poodle,Dalmatian]
  val p: picker.A = potty
  val r: picker.B = picker.choose(potty)(dotty)
  println(r) // dotty
  //type error: println(picker.choose(dotty)(potty))
}
```

We implement `pickLast` as a *meta*-function that explicitly binds a given name. It is not convenient to just return a complex term because then invoking `choose` falls under the term-mem restriction.

val-pickLast[[x_{alt}, T_C, T_A, T_B, e]] = val x_{alt} = new T_{alt} { x_{alt} ⇒ choose: x_{alt}.A →arrow[[x_{alt}.B, x_{alt}.B]] }(
choose (a)=fun[[b, x_{alt}.B, x_{alt}.B, b]]) ;
e

where T_{alt} = choices.Alt$_c$ { x_{alt} ⇒ C: T_C .. T_C, A: T_A .. T_A, B: T_B .. T_B }

3.1 Typechecking Properties

Given
val potty = new pets.Poodle$_c$()
val dotty = new pets.Dalmatian$_c$()

true
val-pickLast[[alt, pets.Dog$_c$, pets.Poodle$_c$, pets.Dalmatian$_c$,
cast[[⊤,
cast[[choices.Alt$_c$,
cast[[choices.Alt$_c$ { alt ⇒ C: ⊥ .. pets.Dog$_c$ },
alt]]]]]]]

true
val-pickLast[[alt, pets.Dog$_c$, pets.Poodle$_c$, pets.Dalmatian$_c$,
cast[[⊤, cast[[alt.A, potty]]]]]

true
cast[[⊤, val-pickLast[[alt, pets.Dog$_c$, pets.Poodle$_c$, pets.Dalmatian$_c$,
cast[[pets.Dalmatian$_c$, app[[alt.choose(potty), dotty]]]]]]]

false
cast[[⊤, val-pickLast[[alt, pets.Dog$_c$, pets.Poodle$_c$, pets.Dalmatian$_c$,
app[[alt.choose(dotty), potty]]]]]

4 F-bounded Abstraction

```
object metachoices {
  trait MetaAlt extends choices.Alt {
    type C = MetaAlt
    type A = C
```

```
    type B = C
  }
  val first = new MetaAlt {
    val choose: C => C => C = a => b => a
    override def toString = "<first>"
  }
  val last = new MetaAlt {
    val choose: C => C => C = a => b => b
    override def toString = "<last>"
  }
  val recfirst = new MetaAlt {
    val choose: C => C => C = a => b => a.choose(a)(b)
    override def toString = "<recfirst>"
  }
  val reclast = new MetaAlt {
    val choose: C => C => C = a => b => b.choose(a)(b)
    override def toString = "<reclast>"
  }
}
```

```
val metachoices = new ⊤{ z ⇒
                         MetaAlt_c: ⊥ .. choices.Alt_c { alt ⇒
                                           C: z.MetaAlt_c .. z.MetaAlt_c,
                                           A: alt.C .. alt.C,
                                           B: alt.C .. alt.C } }()
val mcfirst = new metachoices.MetaAlt_c(
                  choose (a )=fun[[b, metachoices.MetaAlt_c, metachoices.MetaAlt_c, a]])
val mclast = new metachoices.MetaAlt_c(
                  choose (a )=fun[[b, metachoices.MetaAlt_c, metachoices.MetaAlt_c, b]])
val mcrecfirst = new metachoices.MetaAlt_c(
                      choose (a )=fun[[b, metachoices.MetaAlt_c, metachoices.MetaAlt_c,
                                          app[[ a.choose(a), b]]]])
val mcreclast = new metachoices.MetaAlt_c(
                      choose (a )=fun[[b, metachoices.MetaAlt_c, metachoices.MetaAlt_c,
                                          app[[ b.choose(a), b]]]])
```

4.1 Typechecking Properties

```
true
cast[[⊤, app[[ mcrecfirst.choose(mcfirst), mcreclast]]]]
```