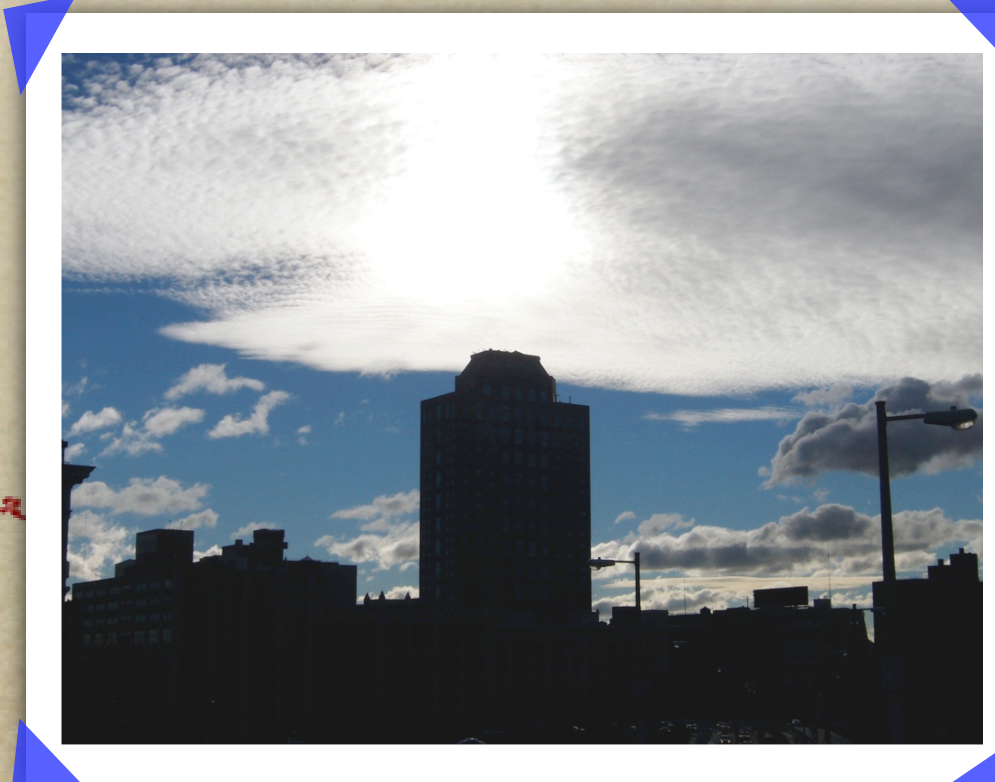
Datatype-Generic Programming

(based on Jeremy Gibbons's work)

27 april 2006,
Spring School on Datatype-Generic Programming,
Nottingham

Datatype-Generic Programming in Scala



Adriaan Moors
adriaan --> cs||kuleuven||be
http://www.cs.kuleuven.be/~adriaan
(based on Jeremy Gibbons's work)

27 april 2006,
Spring School on Datatype-Generic Programming,
Nottingham

Overview

- *Review: DGP in Haskell*
  - *Simple examples*
- *Scala Encoding*
  - *Straightforward OO-encoding*
  - *Simulating equirecursive typing*
- *(Expression problem + GADT in Scala)*

Review: DGP in Haskell (almost)

```
type Fix s a = s a (Fix s a)


class Bifunctor s where
  bimap :: (ba->bc)->(bb->bd)
                      ->(s ba bb)->(s bc bd)

fold :: Bifunctor s => (s fa fb -> fb) -> Fix s fa -> fb
fold f = f . bimap id (fold f)


unfold :: Bifunctor s => (ub -> s ua ub) -> ub -> Fix s ua
unfold f = bimap id (unfold f) . f

dgmap :: Bifunctor s => (ma -> mb) -> Fix s ma -> Fix s mb
dgmap f = bimap f (dgmap f)
-- alternative: dgmap f = fold (bimap f id)
```

Some examples

```
data ListF a b = Nil | Cons a b
        -- a = element type
        -- b = type of recursive substructure

instance Bifunctor ListF where
  bimap f g Nil = Nil
  bimap f g (Cons x y) = Cons (f x) (g y)

type List a = List' a (ListF a)
```

Some examples

```
-- sumall (ConsF 1 (ConsF 2 (ConsF 3 NilF)))
sumall :: Fix ListF Integer -> Integer
sumall = fold myadd
```

```
fold :: Bifunctor s => (s fa fb -> fb) -> Fix s fa -> fb
fold f = f . bimap id (fold f)
```

```
myadd :: ListF Integer Integer ->  Integer
myadd NilF = 0
myadd (ConsF x y) = x+y
```

```
-- preds 10
preds ::  Integer -> List Integer
preds = unfold mypred
```

```
unfold :: Bifunctor s => (ub -> s ua ub) -> ub
                                          -> Fix s ua
unfold f = bimap id (unfold f) . f
```

```
mypred :: Integer ->  ListF Integer Integer
mypred 0 = NilF
mypred x = ConsF (x-1) (x-1)
```

```
dgmap (* 3) (Cons 1 (Cons 2 (Cons 3 Nil)))      -->  Cons 3 (Cons 6 (Cons 9 Nil))
```

Some examples

```
-- sumall (ConsF 1 (ConsF 2 (ConsF 3 NilF)))
sumall :: Fix ListF Integer -> Integer
sumall = fold myadd

myadd :: ListF Integer Integer -> Integer
myadd NilF = 0
myadd (ConsF x y) = x+y
```

```
fold :: Bifunctor s => (s fa fb -> fb) -> Fix s fa -> fb
fold f = f . bimap id (fold f)
```

need more machinery to make this generic too (e.g. PolyP)

```
-- preds 10
preds :: Integer -> List Integer
preds = unfold mypred

mypred :: Integer -> ListF Integer Integer
mypred 0 = NilF
mypred x = ConsF (x-1) (x-1)
```

```
unfold :: Bifunctor s => (ub -> s ua ub) -> ub
                                         -> Fix s ua

unfold f = bimap id (unfold f) . f
```

```
dgmap (* 3) (Cons 1 (Cons 2 (Cons 3 Nil)))     --> Cons 3 (Cons 6 (Cons 9 Nil))
```

Implementation in Haskell

```haskell
data Fix s fa = In (s fa (Fix s fa))
out :: Fix s fa -> s fa (Fix s fa)
out (In x) = x

class Bifunctor s where
    bimap :: (ba->bc)->(bb->bd)->
                  (s ba bb)->(s bc bd)

data ListF la lb = NilF | ConsF la lb
type List a = Fix ListF a
-- instead of: type List a = ListF a (List a)

instance Bifunctor ListF where
    bimap f g NilF = NilF
    bimap f g (ConsF x y) = ConsF (f x) (g y)
```

Implementation in Haskell

```
data Fix s fa = In (s fa (Fix s fa))
out :: Fix s fa -> s fa (Fix s fa)
out (In x) = x

class Bifunctor s where
  bimap :: (ba->bc)->(bb->bd)->
               (s ba bb)->(s bc bd)

data ListF la lb = NilF | ConsF la lb
type List a = Fix ListF a
-- instead of: type List a = ListF a (List a)

instance Bifunctor ListF where
  bimap f g NilF = NilF
  bimap f g (ConsF x y) = ConsF (f x) (g y)
```

Instead of recursive type synonym, make isomorphism

Implementation in Haskell

```
dgmap :: Bifunctor s => (ma->mb) -> Fix s ma -> Fix s mb
dgmap f = In . bimap f (dgmap f) . out

fold :: Bifunctor s => (s foa fob -> fob) -> Fix s foa -> fob
fold f = f . bimap id (fold f ) . out

unfold :: Bifunctor s =>  (ub -> s ua ub) -> ub -> Fix s ua
unfold f = In . bimap id (unfold f ) . f

-- combines an unfold and a fold
hylo :: Bifunctor s =>  (hb -> s ha hb) -> (s ha hc -> hc) -> hb -> hc
hylo f g = g . bimap id (hylo f g) . f

 -- new, not needed before (inserts the In-constructor where f's argument used to be):
build :: Bifunctor s =>  (forall b. (s bua b -> b) -> b) -> Fix s bua
build f = f In
```

Scala?

- *strict, impure, functional OO language*
  - *higher-order functions,*
  - *pattern matching,*
  - *ADT-like class hierarchies (case classes),*
  - *bounded abstract type members,*
  - *type application,*
  - *implicit parameters,*
  - *user-defined coercions (e.g. laziness without syntactic overhead),*
- *http://scala.epfl.ch/ -- http://scala.sygneca.com/*

Scala encoding

- *Stay as close to Haskell code as possible*
- *Ignore laziness* *(in fact, can add it quite elegantly, almost…)*
- *Try to promote argument to* this-status *(when reasonable/possible) -- make it look OO*

Scala code

```
trait TypeConstructor {type a; type b}
// a=type of content, b=type of recursive substructure

trait Bifunctor[s <: Bifunctor[s]] requires s extends TypeConstructor {
// 'this' plays the role of the parameter of type 's a b' in the Haskell code
                                              // type refinement = type app.
   def bimap[c, d](f :a=>c, g :b=>d) :s{type a=c; type b=d}
}

case class Fix[s <: Bifunctor[s], fa](out :s{type a=fa; type b=Fix[s,fa]}) {
   def map[mb](f :fa=>mb) :Fix[s,mb] = Fix(out.bimap(f, .map(f)))
   def fold[fob](f :s{type a=fa; type b=fob} => fob) :fob
            = f(out.bimap(id, .fold(f)))
}
```

Scala code

```
def unfold[s <: Bifunctor[s],ua,ub](f :ub => s{type a=ua; type b=ub})
                                   (x : ub) :Fix[s, ua]
            = Fix(f(x).bimap(id, unfold(f)))
def hylo[s <: Bifunctor[s],ha,hb,hc](f :hb => s{type a=ha; type b=hb},
                          g : s{type a=ha; type b=hc} => hc)(x : hb) :hc
            = g(f(x).bimap(id, hylo[s,ha,hb,hc](f,g)))

abstract class Builder[s <: Bifunctor[s], ba] {
// local polymorphism
  def  build[bb](f :cl (s{type a=ba; type b=bb} => bb)) : bb
 }

def build[s <: Bifunctor[s],ba](b :Builder[s, ba]) :Fix[s,ba]
            = b.build(Fix[s,ba])
```

Some examples

```
abstract class ListF extends Bifunctor[ListF]

  case class NilF[la,lb] () extends ListF {
    type a=la; type b=lb
    def bimap[c, d](f :a=>c, g :b=>d) :NilF[c,d] = NilF()
  }

  case class ConsF[la, lb] (x :la, y :lb) extends ListF {
    type a=la; type b=lb
    def bimap[c, d](f :a=>c, g :b=>d) :ConsF[c,d] = ConsF(f(x), g(y))
  }

type List[a] = Fix[ListF, a]
```

Some examples

```
def sumall( intList : Fix[ListF, Int]) : Int = {
  def myadd( intermed : ListF{type a=Int; type b=Int}) : Int
   = intermed match {
     case NilF() => 0
     case ConsF(x,y) => x+y
   }
  intList.fold(myadd)
}
def preds( i :Int) : List[Int] = {
   def mypred( seed :Int)
         : ListF{type a=Int; type b=Int}
    = seed match {
      case 0 => NilF()
      case x => ConsF(x-1, x-1)
    }
   unfold[ListF, Int, Int](mypred)(i)
}
```

Some examples

```
def sumall( intList : Fix[ListF, Int]) : Int = {
  def myadd( intermed : ListF{type a=Int; type b=Int}) : Int
   = intermed match {
     case NilF() => 0
     case ConsF(x,y) => x+y
   }
  intList.fold(myadd)
}
def preds(
   def my
         :
   = see
     case 0 =
     case x => Cons
   }
  unfold[ListF, Int, Int](mypred)(i)
}
```

```
sumall :: Fix ListF Integer -> Integer
sumall = fold myadd
myadd :: ListF Integer Integer ->  Integer
myadd NilF = 0
myadd (ConsF x y) = x+y
```

Some examples

```
def sumall( intList : Fix[I
  def myadd( in                                  ) : Int
   = interm
     case N
     case C
   }
  intList.fold(
}
def preds( i :Int) : List[Int] = {
   def mypred( seed :Int)
         : ListF{type a=Int; type b=Int}
    = seed match {
      case 0 => NilF()
      case x => ConsF(x-1, x-1)
    }
   unfold[ListF, Int, Int](mypred)(i)
}
```

```
preds ::  Integer -> List Integer
preds = unfold mypred
mypred :: Integer ->  ListF Integer Integer
mypred 0 = NilF
mypred x = ConsF (x-1) (x-1)
```

Some examples

```
def sumall( intList : Fix[ListF, Int]) : Int = {
  def myadd( intermed : ListF{type a=Int; type b=Int}) : Int
   = intermed match {
     case NilF() => 0
     case ConsF(x,y) => x+y
   }
  intList.fold(myadd)
}
def preds( i :Int) : List[Int] = {
   def mypred( seed :Int)
         : ListF{type a=Int; type b=Int}
    = seed match {
      case 0 => NilF()
      case x => ConsF(x-1, x-1)
    }
   unfold[ListF, Int, Int](mypred)(i)
}
```

```
def testLists = {
  print(Fix[ListF, Any](NilF()).map( x => x))

  val list123 = Fix[ListF, Int](ConsF(1,
                Fix[ListF, Int](ConsF(2,
                Fix[ListF, Int](ConsF(3,
                Fix[ListF, Int](NilF())))))))

  print(list123.map( x => x*3)  )

  print(sumall(list123))
  print(preds(10))
}
```

Recursive Types Without the Fuss

```
case class Fix[s <: Bifunctor[s], fa](val out :s{type a=fa; type b=Fix[s,fa]}) {
    def map[mb](f :fa=>mb) :Fix[s,mb] = Fix(out.bimap(f, .map(f)))
    def fold[fob](f :s{type a=fa; type b=fob} => fob) :fob = f(out.bimap(id, .fold(f)))
}

def unfold[s <: Bifunctor[s], ua, ub](f :ub => s{type a=ua; type b=ub})(x :ub) :Fix[s, ua]
    = Fix(f(x).bimap(id, unfold(f)))
```

Recursive Types Without the Fuss

```
case class Fix[s <: Bifunctor[s], fa](val out :s{type a=fa; type b=Fix[s,fa]}) {
    def map[mb](f :fa=>mb) :Fix[s,mb] = bimap(f, .map(f))
    def fold[fob](f :s{type a=fa; type b=fob} => fob) :fob = f(bimap(id, .fold(f)))
}

def unfold[s <: Bifunctor[s], ua, ub](f :ub => s{type a=ua; type b=ub})(x :ub) :Fix[s, ua]
    = f(x).bimap(id, unfold(f))

implicit def unroll[s<: Bifunctor[s],ua](v :Fix[s,ua]) :s{type a=ua; type b=Fix[s,ua]} = v.out
implicit def roll[s<: Bifunctor[s],ra](v :s{type a=ra; type b=Fix[s,ra]}) : Fix[s,ra] = Fix(v)
```

Relevant limitations

- *source of limitation*
  - *Scala (language/compiler)*
  - *me (suggestions welcome ;-))*
- *sometimes type inference fails*
  - *e.g.* `hylo ... = (f(x).bimap(id, hylo[s,ha,hb,hc](f,g)))`
- *sometimes coercions fail*
  - *when target is (implicitly) this (bug)*
  - *no chaining ("feature"...)*
  - `val list123 :Fix[ListF, Int] = ConsF(1, ConsF(2,  ConsF(3,  NilF())))`

GADT's

```
abstract class Expr[t]
  case class Num(num :Int)                                       extends Expr[Int]
  case class Plus(left :Expr[Int], right :Expr[Int])             extends Expr[Int]
  case class Eq(left :Expr[Int], right :Expr[Int])               extends Expr[Boolean]
  case class If[a](c :Expr[Boolean], t :Expr[a], f :Expr[a])     extends Expr[a]
  case class Str(str :String)                                    extends Expr[String]

def eval[t](expr :Expr[t]) :t = expr match {
  case Num(n)      => n
  case Plus(l,r)   => eval(l) + eval(r)
  case Eq(l,r)     => eval(l) == eval(r)
  case If(c, t, f) => if(eval(c)) eval(t) else eval(f)
  case Str(s)      => s
}
```

```
val x = If ( Eq(Num(1), Num(2)), Str("one==two!"), Str("phew..."))
val s :String = eval(x)
```

Independently Extensible Solutions to the Expression Problem

Matthias Zenger, Martin Odersky

École Polytechnique Fédérale de Lausanne
INR Ecublens
1015 Lausanne, Switzerland

Technical Report IC/2004/33

Abstract

The *expression problem* is fundamental for the development of extensible software. Many (partial) solutions to this important problem have been proposed in the past. None of these approaches solves the problem of using different, independent extensions jointly. This paper proposes solutions to the expression problem that make it possible to combine independent extensions in a flexible, modular, and type-safe way. The solutions, formulated in the programming language SCALA, are affected with only a small implementation overhead and are easy to implement by hand.

1 The Expression Problem

Since software evolves over time, it is essential for software systems to be extensible. But the development of extensible software poses many design and implementation problems, especially, if extensions cannot be anticipated. The *expression problem* is probably the most fundamental one among these problems. It arises when recursively defined datatypes and operations on these

challenge is now to find an implementation technique which satisfies the following list of requirements:

- *Extensibility in both dimensions:* It should be possible to add new data variants and adapt existing operations accordingly. Furthermore, it should be possible to introduce new processors.
- *Strong static type safety:* It should be impossible to apply a processor to a data variant which it cannot handle.
- *No modification or duplication*: Existing code should neither be modified nor duplicated.
- *Separate compilation:* Compiling datatype extensions or adding new processors should not encompass re-type-checking the original datatype or existing processors.

We add to this list the following criterion:

- *Independent extensibility:* It should be possible to combine independently developed extensions so that they can be used jointly [21].

Expression Problem

(base functionality 1/2)

```
trait BaseExpr {
    type TExpr <: ExprBase
    type TNum <: NumBase with TExpr
    type TPlus <: PlusBase with TExpr
    type TEq <: EqBase with TExpr
    type TIf <: IfBase with TExpr

    abstract class ExprBase requires TExpr {
        type t
        def eval :t
    }

    trait NumBase requires TNum extends ExprBase {
        type t = Int
        val n :Int
        def eval :t = n
    }
```

Expression Problem

(base functionality 2/2)

```
trait PlusBase requires TPlus extends ExprBase {
  type t = Int
  val l :TExpr{type t=Int}
  val r :TExpr{type t=Int}
  def eval :t = l.eval + r.eval
}
trait EqBase requires TEq  extends ExprBase {
  type t = Boolean
  val l :TExpr{type t=Int}
  val r :TExpr{type t=Int}
  def eval :t = l.eval == r.eval
}
trait IfBase  requires TIf  extends ExprBase {
  val c :TExpr{type t=Boolean}
  val bTrue :TExpr{type t=IfBase.this.t}
  val bFalse :TExpr{type t=IfBase.this.t}
  def eval :t = if(c.eval) bTrue.eval else bFalse.eval
}
}
```

New datatype

```
trait StringExpr extends BaseExpr {
  type TString <: StringBase with TExpr

  trait StringBase requires TString extends ExprBase {
    type t = String
    val s :String

    def eval :t = s
  }
}
```

New functionality

```
trait ShowExpr extends BaseExpr {
 type TExpr <: ExprBase with ExprShow

   trait ExprShow requires TExpr extends ExprBase {
      def show :String = toString() + " = " + eval
   }
}
```

Putting it all together

```
object main extends BaseExpr with ShowExpr with StringExpr {
  type TExpr = Expr
  type TNum = Num
  type TPlus = Plus
  type TEq = Eq
  type TIf = Expr with IfBase  // because couldn't write type TIf[a] <: TIfBase[a]
  type TString = StringExpr

  abstract class Expr extends ExprBase with ExprShow
    case class Num(num :Int) extends Expr with NumBase {val n=num}
    case class Plus(left :TExpr{type t=Int}, right :TExpr{type t=Int}) extends Expr with PlusBase
                    {val l=left; val r=right}
    case class Eq(left :TExpr{type t=Int}, right :TExpr{type t=Int})  extends Expr with EqBase
                    {val l=left; val r=right}
    case class If[a](cond :TExpr{type t=Boolean}, bT :TExpr{type t=a}, bF :TExpr{type t=a})
          extends Expr with IfBase {type t=a; val c=cond; val bTrue=bT; val bFalse = bF}
    case class StringExpr(str :String)  extends Expr with StringBase {val s=str}
```

Execute!

```
def main(args :Array[String]) :Unit = {
  val x = If ( Eq(Num(1), Num(2)), Str("one==two!"), Str("phew..."))
  val s :String = x.eval
  System.out.println(x.show )
 }
}
```

That’s all, folks!

- Thank you for your attention!
- Questions?

- *contact me at: adriaan<>cs,kuleuven,be*
- *full code: http://www.cs.kuleuven.be/~adriaan*