Datatype-Generic Programming with First-Class Regular Functors

(Finally some type system hacking, yeah!)

Cai Yufei, Paolo G. Giarrusso, Klaus Ostermann
University of Tübingen, Germany

Context

- Algorithms on complex data structures are often repetitive and fragile
  - They depend on details of the data structure that are not relevant to the algorithm
  - The same traversal/recursion scheme is repeated in many algorithms

Example

```
data Term = Var String | Lam String Term
          | App Term Term | Lit Int


rename :: Term -> (String -> String) -> Term
rename (Var x) f = Var (f x)
rename (Lam x t) f = Lam (f x) (rename t f)
rename (App t1 t2) f = App (rename t1 f) (rename t2 f)
rename (Lit n) f = Lit n
```

Algorithm is only interested in names but is coupled to the full structure of terms

Structural recursion on terms will be replicated in many algorithms

Datatype-Generic Programming

- Hard to define precisely (but Gibbons tried)
- Deals with “these problems”
- Many previous approaches
  - Bananas, Lenses & Barbed Wire, Origami
  - PolyP, Generic Haskell
  - Scrap Your Boilerplate, Strafunski
- Ad-hoc vs. parametric datatype genericity
  - We are shooting for parametric genericity

Our elevator pitch

- Datatypes can be described via functors
- Functors can
  - Define recursion schemes
  - Define a “view” on a data structure as a container
- Main insight
  - The same datatype can be defined in many different ways via functors
  - We can use functors as an extensible set of “views” on a datatype
  - These views can be used to decouple algorithms from the shape and recursion structure of the data
- Functors first!
  - Datatypes derived from functors and not the other way around
- We have implemented a Scala (macro) library, Creg, that implements the idea

What is a functor?

- For the purpose of this talk a functor is a type constructor F together with a "map" function such that *blah* (I'll rather show the code)

```
trait Functor {
  type Map[+X]
  def fmap[A,B](f: A => B) : Map[A] => Map[B]
}
```

- Standard algebraic data types can be understood as least fixed points of polynomial functors. Example: Integer lists are the least FP of F(X) = 1 + Int * X

What is a regular functor?

- Polynomial functors + a built-in fixed point constructor
- E.g. List[X] = Fix(Z -> 1+X*Z)
- Datatypes are just fully applied regular functors

Known properties of regular functors

- Can describe algebraic datatype
- Standard recursion schemes can be defined in terms of fmap
  - Catamorphisms (“folds”), Anamorphisms (“unfolds”), …

```
def cata[T](F: Traversable)(f: F.Map[T] => T): Fix[F.Map] => T =
  xs => f( F.fmap(cata(F)(f),xs.unroll))
```

- Can be used to derive a very generic traverse function

```
def traverse[A, B](G: Applicative)(f: A => G.Map[B]):  Map[A] =>  Map[Map[B]]
```

One datatype, multiple views

```
@functor def termF[term] =
  TermT {
    Lit(value = Int)
    Var(name = String)
    Abs(param = String, body = term)
    App(op = term, arg = term)}
```

```
@functor def nameF[tau] = Fix(term =>
  TermT {
    Lit(value = Int)
    Var(name = tau)
    Abs(param = tau, body = term)
    App(op = term, arg = term)}})
```

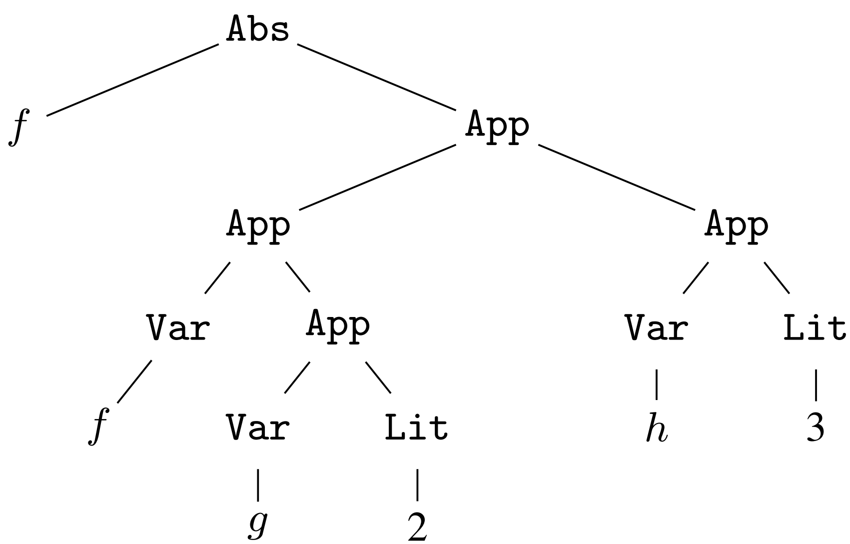
Term = Fix(termF) = nameF(String)

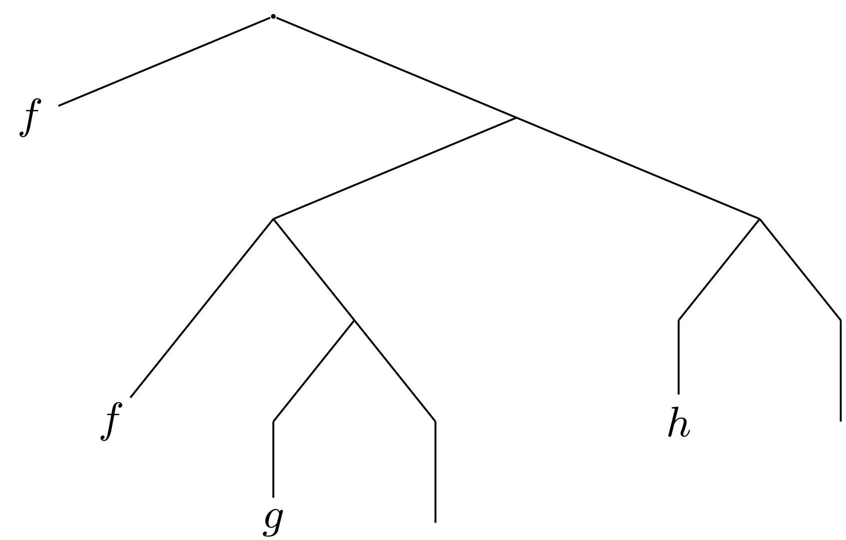
We can choose the functor (and hence structure & recursion schemes) that is best for the algorithm at hand!

One datatype, multiple views

```
@functor def termF[term] =
  TermT {
    Lit(value = Int)
    Var(name = String)
    Abs(param = String, body = term)
    App(op = term, arg = term)}
```

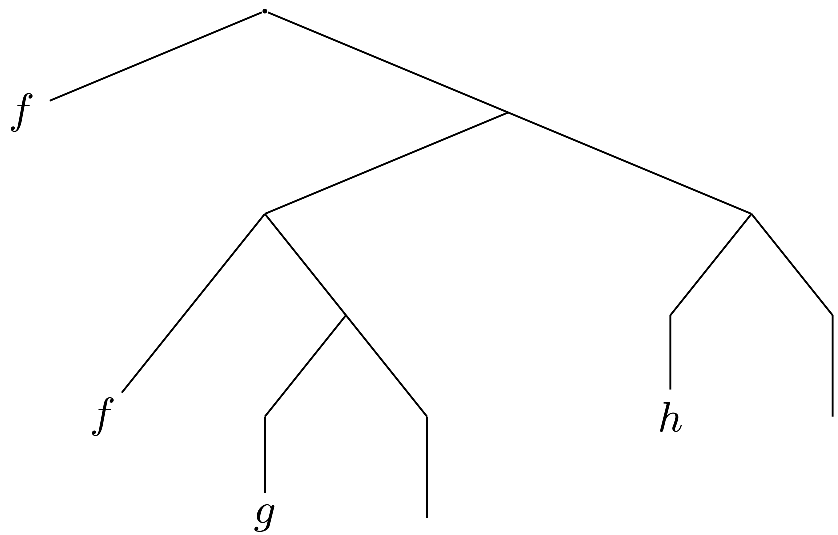
```
@functor def nameF[tau] = Fix(term =>
  TermT {
    Lit(value = Int)
    Var(name = tau)
    Abs(param = tau, body = term)
    App(op = term, arg = term)}))
```





We can choose the functor (and hence structure & recursion schemes) that is best for the algorithm at hand!

One datatype, multiple views




```
@functor def nameF[tau] = Fix(term =>
  TermT {
    Lit(value = Int)
    Var(name = tau)
    Abs(param = tau, body = term)
    App(op = term, arg = term)}))
```

```
def rename(t: Term, f: String => String): Term = nameF.fmap[String,String](t,f)
```

Algorithm decoupled from structure of terms

Using derived recursion schemes and generic traversals

Derived from fmap and termF

```
def count(t : Term)= cata[Int](t) {
  case Lit(n) => 1
  case other  => termF(other).reduce(0, _ + _)
}
```

Derived from traverse

Such generic traversals are not new.
But more flexibility:
generic traversals, catamorphisms etc. are now available for every functor!

One datatype, multiple recursion schemes

```
@functor def opF[t] = TermT {
  Lit(value = Int)
  Var(param = String)
  Abs(param = String, body = Term)
  App(op = t, arg = Term)
}
```

Recurse only into operator position!

```
def getOperator(t: Term): Term =
 cata(opF)({
   case App(op, arg) => op
   case operator     => operator
  })(t)
```

Find left-most operator (*)

(*) this code is a (small) lie due to our iso-recursive encoding of datatypes

Alas…

Term = Fix(termF) = termF(Fix(termF)) = nameF[String] = Fix(opF)

How it works (1/3)

```
@structure def TermT = {
  Lit(value) ;
  Var(name) ;
  Abs(param, body) ;
  App(op, arg) }
```

```
sealed trait TermT[L, V, Ab, Ap]

case class Lit[I](value: I)
  extends TermT[Lit[I], ⊥, ⊥, ⊥]

case class Var[S](name: S)
  extends TermT[⊥, Var[S], ⊥, ⊥]

case class Abs[S, T](param: S, body: T)
  extends TermT[⊥, ⊥, Abs[S, T], ⊥]

case class App[T1, T2](op: T1, arg: T2)
  extends TermT[⊥, ⊥, ⊥, App[T1, T2]]
```

Declaration of nominal products and sums

Declared separately to ensure interoperability of functors

How it works (2/3)

Functors are values!

```
@functor def termF[term] =
  TermT {
    Lit(value = Int)
    Var(name = String)
    Abs(param = String, body = term)
    App(op = term, arg = term)}
```

```
val termF = new Traversable {
  type Map[+T] =
    TermT[Lit[Int],
          Var[String],
          Abs[String, T],
          App[T, T]]
  def traverse ...
  def fmap ...
  def apply[T](x: Map[T]) ...
}
```

```
@data Term = Fix(termF)
```

```
type Term = Fix[termF.Map]
```

```
sealed trait Fix[+F[+_]] {
  def unroll: F[Fix[F]] }
```

library code

Example:

Lit(5) : TermT[Lit[Int], ⊥, ⊥, ⊥] <: termF.Map[Term]

new Fix[termF.Map]{ def unroll = Lit(5) } : Term

iso-recursive types ☹

How it works (3/3)

```
@functor def nameF[tau] = Fix(term =>
 TermT {
  Lit(value = Int)
  Var(name = tau)
  Abs(param = tau, body = term)
  App(op = term, arg = term)
} )
```

```
val nameF = new Traversable {
 type Map[+tau] =
  Fix[F[tau]#λ ]
 private[this] type F[+tau]
  = { type λ[+T] = TermT[
    Lit[Int], Var[tau], Abs[tau, T], App[T, T]]}
 def traverse ...
 def fmap ...
}
```

```
val t1 : Term = ...
val t2 : nameF.Map[String] = t1
```

Isomorphisms across recursion schemes

```
@functor def opF[t] = TermT {
  Lit(value = Int)
  Var(param = String)
  Abs(param = String, body = Term)
  App(op = t, arg = Term)
}
```

```
val opF = new Traversable {
  type Map[+T] = TermT[
      Lit[Int], Var[String],
      Abs[String, Term],
      App[T, Term]]}
  def traverse ...
  def fmap ...
}
```

```
val t1 : Term = ...
val t2 : Fix[opF.Map] = coerce(t1)
```

Magic involving roll/unroll of the respective Fix applications

Related Work: Generic Haskell

(and PolyP is similar)

$$
\begin{aligned}
&\textbf{type}\ Encode\{[*]\} \quad t = t \rightarrow [Bool] \\
&\textbf{type}\ Encode\{[k \rightarrow l]\}\ t = \forall a.\ Encode\{[k]\}\ a \rightarrow Encode\{[l]\}\ (t\ a)
\end{aligned}
$$

$$
\begin{aligned}
&encode\{|t :: k|\} && :: Encode\{[k]\}\ t \\
&encode\{|Char|\}\ c && = encodeChar\ c \\
&encode\{|Int|\}\ n && = encodeInt\ n \\
&encode\{|Unit|\}\ unit && = [\,] \\
&encode\{|{:+:}|\}\ ena\ enb\ (Inl\ a) && = False : ena\ a \\
&encode\{|{:+:}|\}\ ena\ enb\ (Inr\ b) && = True : enb\ b \\
&encode\{|{:\times:}|\}\ ena\ enb\ (a :\times: b) && = ena\ a +\!\!+\ enb\ b
\end{aligned}
$$

We cannot (easily) define polytypic functions because we use nominal sums and products. Also, we cannot pattern-match on the structure of types (parametric vs. ad-hoc datatype genericity)

Functors are derived from datatypes and not the other way around

Related Work: Scrap Your Boilerplate

- Provides functions like:

```
gmapQ :: ∀a . Data a ⇒ (∀b . Data b ⇒ b → c) → a → [ c ].
```

- Can generically define operations on all occurences of a type in an ADT
- Using Scala's TypeTags, we can encode gfoldl and friends
- We can be more fine-grained than SYB
  – Not all occurences of a type are necessarily treated the same

More related work

- Can encode Origami [Gibbons]
- Can encode compos [Bringert & Ranta]
- There's tons of additional related work…

Questions?