Revisiting Catamorphisms over Datatypes with Embedded Functions

Leonidas Fegaras Tim Sheard

Oregon Graduate Institute

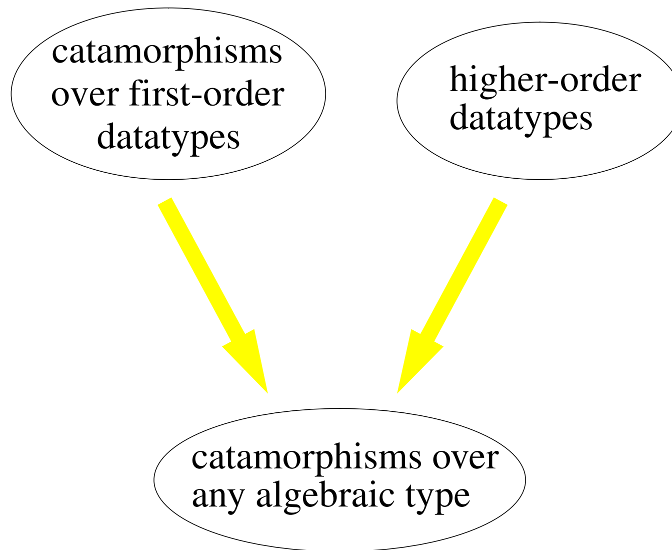
What are catamorphisms?

- A family of functions: often called *fold* operators.
- They replace data constructors with arbitrary functions.

Catamorphisms are useful:

- they are elegant means of expressing algorithms;
- they facilitate reasoning about programs;
- they support program optmization (e.g., deforestation).

Goal of this paper



Earlier work: Paterson [draft'94], and Meijer & Hutton [FPCA'95]

The list catamorphism

Similar to ML's fold and Haskell's foldr.

datatype List(α) = Nil | Cons **of** $\alpha \times$ List(α)

```
fun cata(n,c) Nil           = n
  | cata(n,c) (Cons(a,r)) = c( a, cata(n,c) r )
```

Function c replaces Cons, value n replaces Nil:

cata(n,c) (Cons(1,Cons(2,Cons(3,Nil)))) = c(1,c(2,c(3,n)))

Example

```
fun filter(p) x =
      cata( Nil, λ(a,r). if p(a) then Cons(a,r) else r ) x
            n    c
```

filter(even) (Cons(1,Cons(2,Cons(3,Cons(4,Nil)))))

= c(1,c(2,c(3,c(4,n))))

= Cons(2,Cons(4,Nil))

Catamorphisms can be easily defined for most first-order algebraic datatype.

```
datatype Term = Const of int
              | Var of string
              | Appl of Term × Term
              | Abs of string × Term
```

(λx. x) 5 → Appl(Abs("x",Var "x"),Const 5)

```
fun cataT(c,v,p,a) (Const n)   = c n
  | cataT(c,v,p,a) (Var s)     = v s
  | cataT(c,v,p,a) (Appl(x,y)) = p( cataT(c,v,p,a) x,
                                    cataT(c,v,p,a) y )
  | cataT(c,v,p,a) (Abs(s,e))  = a( s, cataT(c,v,p,a) e )
```

Higher-order datatypes

No problem, unless they contain a *contravariant recursion*:

datatype T = C **of** int | D **of** T → T

Why bother? There are 3 examples:

- higher-order abstract syntax;
- circular lists;
- graphs.

Higher-order abstract syntax

Pfenning & Elliot [*PLDI'88*], Wand [*POPL'93*], Despeyroux & Hirschowitz [*LPAR'94*]

Closed terms of λ–calculus:

```
datatype Term = Const of int
              | Appl of Term × Term
              | Abs of Term → Term
```

(λx. x x) 5 → Appl(Abs(**fn** x ⇒ Appl(x,x)),Const 5)

Evaluator over higher-order syntax

datatype Term = Const **of** int
| Appl **of** Term $\times$ Term
| Abs **of** Term $\rightarrow$ Term

datatype Value = Num **of** int | Fun **of** Value$\rightarrow$Value

eval: Term $\rightarrow$ Value

fun eval(Const n) = Num n
| eval(Appl(f,e))= **let val** Fun(g) = eval(f)
in g(eval(e)) **end**
| eval(Abs f) = Fun(…)

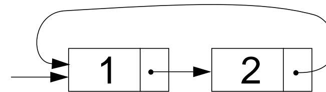
eval(Appl(Abs(**fn** x $\Rightarrow$ x),Const 1)) = Num 1

Circular Lists

datatype Clist(α) = Nil
| Cons **of** $\alpha \times$ Clist(α)
| Rec **of** Clist(α) $\rightarrow$ Clist(α)

fun head(Cons(a,r)) = a
| head(Rec f) = head(f(Rec f))

The circular list [1,2,1,2,1,2,...] is:



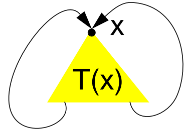
Rec(**fn** x $\Rightarrow$ Cons(1,Cons(2,x)))

[1,2,3,4,...] is:

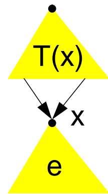
Rec(**fn** x $\Rightarrow$ Cons(1, map(**fn** z $\Rightarrow$ z+1) x))

Graphs

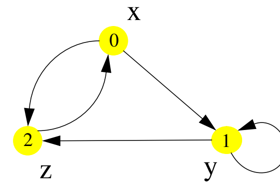
datatype Graph(α) = Node **of** $\alpha \times$ List(Graph(α))
| Rec **of** Graph(α) $\rightarrow$ Graph(α)
| Share **of** (Graph(α) $\rightarrow$ Graph(α)) $\times$ Graph(α)



Rec(**fn** x $\Rightarrow$ T(x))
$\cong$ Y(T)



Share(**fn** x $\Rightarrow$ T(x), e)
$\cong$ **let val** x = e
in T(x) **end**



fun root(Node(a,r)) = a
| root(Rec f) = root(f(Rec f))
| root(Share(f,e)) = root(f e)

Rec(**fn** x $\Rightarrow$ Share(**fn** z $\Rightarrow$ Node(0,[z,Rec(**fn** y $\Rightarrow$ Node(1,[y,z]))]),
Node(2,[x])))

An example of our technique

- the recursive Term evaluator in the style of Paterson-Meijer-Hutton;
- the problems of this method;
- the recursive Term evaluator in our style;
- the catamorphism over Term.

The λ term evaluator (à la Paterson-Meijer-Hutton)

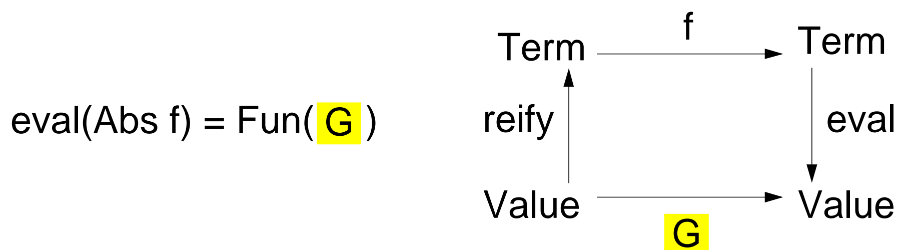
datatype Term = Const **of** int
 | Appl **of** Term $\times$ Term
 | Abs **of** Term $\rightarrow$ Term

datatype Value = Num **of** int | Fun **of** Value$\rightarrow$Value

fun eval(Const n) = Num n
| eval(Appl(f,e))= **let val** Fun(g) = eval(f)
 in g(eval(e)) **end**
| eval(Abs f) = Fun(G)

Need to define G : Value $\rightarrow$ Value

eval(Abs f) = Fun(G)



fun eval(Const n) = Num n
| eval(Appl(f,e))= **let val** Fun(g) = eval(f) **in** g(eval(e)) **end**
| eval(Abs f) = Fun(eval $\circ$ f $\circ$ reify)

and reify(Num n) = Const n
| reify(Fun g) = Abs(reify $\circ$ g $\circ$ eval)

Evaluating (λx. x) 1 using the eval-reify approach

eval(Appl(Abs(**fn** x ⇒ x),Const 1))

= **let val** Fun(g) = eval(Abs(**fn** x ⇒ x))
in g(eval(Const 1)) **end**

= **let val** Fun(g) = Fun(**fn** x ⇒ eval(reify(x)))
in g(eval(Const 1)) **end**

= eval(reify(eval(Const 1)))

= eval(reify(Num 1))

= eval(Const 1)

= Num 1

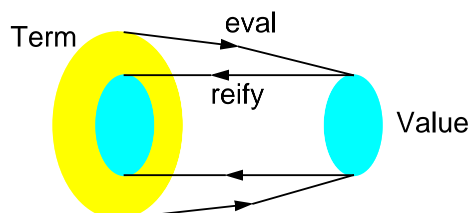
Problems of previous work

- Double the effort: eval needs reify, print needs parse;
- Redundancy of computation:

  eval(Appl(Abs f,e)) = eval(f(reify(eval(e))))

- reify must be the right inverse of eval:

  ∀x∈ *range*(eval): eval(reify(x)) = x



eval(Abs(**fn** x ⇒ x))
= Fun(eval ° (**fn** x ⇒ x) ° reify)
= Fun(**fn** x ⇒ eval(reify x))
= Fun(**fn** x ⇒ x)

Our approach: Using an extra constructor Place to shortcut reify

```
datatype Term =  Const of int
               | Appl of Term × Term
               | Abs of Term → Term
               | Place of Value

fun eval(Const n)     =  Num n
  | eval(Appl(f,e))   =  let val Fun(g) = eval(f) in g(eval(e)) end
  | eval(Abs f)       =  Fun( eval ° f ° Place )
  | eval(Place x)     =  x
```

Place satisfies the desired property: eval(Place x) = x

Evaluating (λx. x) 1 using the Place constructor

```
eval( Appl(Abs(fn x ⇒ x),Const 1) )

= let val Fun(g) = eval(Abs(fn x ⇒ x))
  in g(eval(Const 1)) end

= let val Fun(g) = Fun(fn x ⇒ eval(Place(x)))
  in g(eval(Const 1)) end

= eval(Place( eval(Const 1) ))

= eval(Place(Num 1))

= Num 1
```

The catamorphism over Term

```
datatype Term(α) = Const of int
                 | Appl of Term(α) × Term(α)
                 | Abs of Term(α) → Term(α)
                 | Place of α

fun cataT(c,p,a) (Const n) = c n
  | cataT(c,p,a) (Appl(x,y))= p( cataT(c,p,a) x, cataT(c,p,a) y )
  | cataT(c,p,a) (Abs f)    = a( cataT(c,p,a) ∘ f ∘ Place )
  | cataT(c,p,a) (Place x)  = x

fun eval x  =  cataT( Num, fn (Fun(f),e) ⇒ f e, Fun ) x

eval: Term(Value) → Value
```


Restrictions

To avoid problems:

- The Place constructor should only be used by a cata.

  Solution: make Place and cata primitives and hide Place from users.

- A contravariant variable should not be analyzed.

  eval(Abs(**fn** x ⇒ **case** x **of** ...)) → Fun(**fn** x ⇒ eval(**case** (Place x) **of** ...))

  eval(Abs(**fn** x ⇒ cataT(...) x)) → Fun(**fn** x ⇒ eval(cataT(...)(Place x))

  Solution: use the type system to detect these cases.

Doing this in general: the catamorphism over any algebraic type T

It is defined in terms of the *functor* of T.

Some background

The *functor* for $\tau(\alpha)$ maps a function f into the function $[\![\tau(\alpha)]\!](f)$:

$$f : t_1 \to t_2 \quad \Rightarrow \quad [\![\tau(\alpha)]\!](f) : \tau(t_1) \to \tau(t_2)$$

$$\begin{aligned}
[\![\alpha]\!](f) &= f \\
[\![\tau_1 \times \tau_2]\!](f) &= [\![\tau_1]\!](f) \times [\![\tau_2]\!](f) \\
[\![T(\tau)]\!](f) &= map^T([\![\tau]\!](f)) \\
[\![\tau]\!](f) &= id
\end{aligned}$$

Notation:

$f \times g = \lambda(x,y).\ (\ f\ x,\ g\ y\ )$

$id = \lambda x.\ x$

e.g., $[\![\ \mathsf{int} \times \alpha \times \mathsf{list}(\alpha)\ ]\!](f) = id \times f \times map(f)$
$= \lambda(x,y,z).\ (\ x,\ f(y),\ map(f)\ z\ )$

Properties: $[\![\tau]\!](id) = id$ $\quad [\![\tau]\!](f) \circ [\![\tau]\!](h) = [\![\tau]\!](\ f \circ h\ )$

Extending functors to arrow types

Need to differentiate positive from negative types:

$$\underbrace{(\overset{+}{\text{int}} \rightarrow \overset{-}{\text{bool}})}_{-} \rightarrow \overset{+}{\text{int}}$$

Positive and negative instances of a type variable α:

$$\textbf{type}\ T(\alpha) \;=\; \underbrace{(\overset{+}{\alpha} \rightarrow \overset{-}{\alpha})}_{-} \rightarrow \underbrace{(\overset{-}{\alpha} \rightarrow \overset{+}{\alpha})}_{+}$$

Bifunctors

The *bifunctor* of a type $\tau(\alpha)$:

$$f: t_1 \rightarrow t_2 \quad \wedge \quad g: t_2 \rightarrow t_1 \quad \Rightarrow \quad [\![\tau(\alpha)]\!](f,g): \tau(t_1) \rightarrow \tau(t_2)$$

$$
\begin{array}{lcl}
[\![\alpha]\!](f,g) & = & f \\
[\![\tau_1 \times \tau_2]\!](f,g) & = & [\![\tau_1]\!](f,g) \times [\![\tau_2]\!](f,g) \\
[\![\tau_1 \rightarrow \tau_2]\!](f,g) & = & \lambda h.\ [\![\tau_2]\!](f,g) \circ h \circ [\![\tau_1]\!](g,f) \\
[\![T(\tau)]\!](f,g) & = & map^T([\![\tau]\!](f,g)) \\
[\![\tau]\!](f,g) & = & id
\end{array}
$$

Properties:

$$
\begin{array}{lcl}
[\![\tau]\!](id,id) & = & id \\
[\![\tau]\!](f,g) \circ [\![\tau]\!](h,k) & = & [\![\tau]\!](f \circ h, k \circ g)
\end{array}
$$

The catamorphism over a type T

Type T has n data constructors $C_i: \tau_i(T) \to T$,
plus one extra constructor $Place^T$:

$$cata^T(f_1,\ldots,f_n) \circ C_i = f_i \circ [\![\tau_i(\alpha)]\!](\, cata^T(f_1,\ldots,f_n), Place^T\,)$$
$$cata^T(f_1,\ldots,f_n) \circ Place^T = id$$

```
                          +          −          +
datatype T(β) = C of ( T(β) → T(β) ) → T(β)
                      | PlaceT of β

fun cataT(f)(C g) = f(cataT(f) ° g ° (fn h ⇒ PlaceT ° h ° cataT(f)))
|     cataT(f) (PlaceT x) = x
```


Conclusion

Simulating the right inverse of a function with an extra data constructor Place is useful for:

- meta-programming;
- circular lists and graphs.

It might also be useful for:

- higher-order unification;
- fusing programs using parametricity laws.

Equality between terms

```
fun eqT(Const(n),Const(m)) k  =  ( n=m )
|   eqT(Appl(a,b),Appl(a',b') k =  eqT(a,a') k ∧ eqT(b,b') k
|   eqT(Abs(f),Abs(g)) k        =  eqT(f(Place k),g(Place k)) (k+1)
|   eqT(Place n,Place m) k      =  ( n=m )
|   eqT _ _                     =  false
```

```
eqT( Abs(fn x ⇒ Appl(x,x)), Abs(fn y ⇒ Appl(y,y)) ) 0
    =  eqT( Appl(Place 0,Place 0), Appl(Place 0,Place 0) ) 1
    =  eqT(Place 0,Place 0) 1 ∧ eqT(Place 0,Place 0) 1
    =  (0=0) ∧ (0=0)
    =  true
```


Program fusion

The Y combinator for functions:

$$Y(f)\ x = f(Y(f))\ x$$

$$Y:\ \forall \alpha,\beta.\ ((\alpha\rightarrow\beta)\rightarrow\alpha\rightarrow\beta)\rightarrow\alpha\rightarrow\beta$$

The parametricity theorem for Y is the unfold-simplify-fold law:

$$\forall g,f,\phi,h:\quad g \circ (f\ h) = \phi\ (g \circ h) \ \Rightarrow\ g \circ (Y\ f) = Y\ \phi$$

To find a solution for ϕ:

1) choose Place such that $g \circ \text{Place} = id$,
2) choose k such that $h = \text{Place} \circ k$, then:

$$g \circ (Y\ f) = Y(\lambda k.\ g \circ (f\ (\text{Place} \circ k)))$$

datatype list(α,β) = Nil | Cons **of** $\alpha \times$ list(α,β) | Place **of** β

```
fun  len Nil           = 0                fun  inc Nil           = Nil
|    len (Cons(a,r))   = 1+(len x)        |    inc (Cons(a,r))   = Cons(a+1,inc r)
|    len (Place x)     = x                |    inc (Place x)     = x
```

f x = len(inc x)

```
fun  f Nil           = len(Nil)
|    f (Cons(a,r))   = len( Cons(a+1,Place(f r)) )
|    f (Place x)     = x
```

f [a_1,…,a_n] creates n garbage Cons-Place cells:
Cons(a_i, Place(n-i))

Solving parametricity equations

The parametricity theorem for the type $\forall\alpha.\ \tau(\alpha)$ that relates x and y is:

$$x = [\![\tau(\alpha)]\!](\ f, \text{Place}\ )\ y$$
$$f \circ \text{Place} = id$$

Example:

g: $\forall\alpha: \tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \alpha$

$f(g\ x_1\ \ldots\ x_n)\ =\ g\ ([\![\tau_1]\!](\ f, \text{Place}\ )\ x_1)\ \ldots\ ([\![\tau_n]\!](\ f, \text{Place}\ )\ x_n)$

cata: $\alpha\times(\beta\times\alpha\rightarrow\alpha)\rightarrow \text{list}(\beta)\rightarrow\alpha$

f(cata(n,c) x) = cata(f(n), λ(a,r). f(c(a,Place r))) x