Boxes Go Bananas: Parametric Higher-Order Abstract Syntax in System F

Stephanie Weirich

University of Pennsylvania

Joint work with Geoff Washburn

Catamorphisms

- Catamorphisms (bananas -- $(\!|\quad|\!)$) are “folds” over datastructures.
  - **`foldr`** on lists is the prototypical catamorphism.
- Many useful operations can be expressed as catamorphisms (**`filter`**, **`map`**, **`flatten`**…).
- Using catamorphisms means that you can reason about programs algebraically.
- Problem: how do we implement catamorphisms over data structures that contain functions?

Overview of talk

- If the functions in the datatype are parametric, then there is an easy way to define the catamorphism.
- Previous work: use a special-purpose type system to guarantee parametricity.
- Today: use Haskell + first-class polymorphism for the same task.
- Nice connections with previous work.

Datatypes with Functions

- Untyped ¸-calculus in Haskell

```
data Exp = Var String
         | Lam String Exp
         | App Exp Exp
```

- With this datatype we need to write tricky code for capture avoiding substitution.
- Alternative: Higher-Order Abstract Syntax (HOAS).

Higher-Order Abstract Syntax

- Old idea – goes back to Church.
- Implement bindings in the object language using meta-language bindings.

```
data Exp = Lam (Exp -> Exp)
         | App Exp Exp
```

- Examples:
  - `Lam(\x -> x)`

  -
    ```
    App (Lam (\x -> App x x))
        (Lam (\x -> App x x))
    ```
- Substitution is function application.

Bananas in Space

- Meijer and Hutton extended classic “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire” to support datatypes with embedded functions, such as HOAS.
- Define catamorphism by simultaneously defining its inverse, the anamorphism.
- Problem: many functions do not have obvious or efficient inverses.
  - Inverse of hash function?
  - Inverse of pretty-print requires parsing.

Bananas in Space

```
data ExpF a = App a a | Lam (a -> a)
data Exp = Roll (ExpF Exp)

app :: Exp -> Exp -> Exp
app x y = Roll (App x y)
lam :: (Exp -> Exp) -> Exp
lam x = Roll (Lam x)

cata :: (ExpF a -> a) -> (a -> ExpF a)
          -> Exp -> a
```

Recursive type is fixed point of ExpF

Use **`ExpF`** in types of args to cata.

Example: Evaluation

```
data Value = Fn (Value -> Value)

eval :: Exp -> Value
eval = cata f g where
  f :: ExpF Value -> Value
  f (App (Fn x) y) = x y
  f (Lam x) = Fn x
  g :: Value -> ExpF Value
  g (Fn x) = Lam x
```

Bananas in Space

```
cata :: (ExpF a -> a) -> (a -> ExpF)
         -> Exp -> a
cata f g (app x y) =
  f (App (cata f g x) (cata f g y))

cata f g (lam x) =
  f (Lam ((cata f g) . x . (ana f g)))

ana :: (ExpF a -> a) -> (a -> ExpF)
         -> a -> Exp
```

```
x :: Exp -> Exp
```

Programs from Outer Space

- If the function is *parametric*, the inverse only undoes work that will be redone later.
- Fegarus & Sheard: don't do the work to begin with.
- Introduce a placeholder:

```
data Exp a = Roll (ExpF (Exp a))
           | Place a
```

- Parameterize Exp with the result type of catamorphism.

Catamorphisms with Place

- Catamorphism

```
cata :: (ExpF a -> a) -> Exp a -> a
cata f (app x y) =
 f (App (cata f x) (cata f y))
cata f (lam x) =
 f (Lam (cata f) . x . Place)
cata f (Place x) = x
```

An Example

```
countvar :: Exp Int -> Int
countvar = cata f

f :: ExpF Int -> Int

f (App x y) = x + y
f (Lam f) = f 1
```

```
x,y :: Int
```

```
f :: Int -> Int
```

Evaluation of `countvar`

```
countvar (lam (\x -> app x x))
= cata f (lam (\x -> app x x))
= f (Lam ((cata f) .
             (\x -> app x x) . Place ))
= ((\x -> cata f (app (Place x)(Place x))
   1)
= cata f (app (Place 1)(Place 1))
= f (App (cata f (Place 1))
         (cata f (Place 1)))
= (cata f (Place 1)) + (cata f (Place 1))
= 1 + 1
= 2
```

Only for parametric datatypes

- Infinite Lists (in an eager language).

```
data IListF a = Cons Int a
              | Mu (a -> a)
cons x y = Roll (Cons x y)
mu x = Roll (Mu x)
```

- List of ones

```
ones = mu (\x -> cons 1 x)
```

- Alternating 1's and 0's

```
onezero = mu (\x -> cons 1 (cons 0 x))
```

Using Infinite Lists

- Catamorphism

```
cata :: (IListF a -> a) -> IList a -> a
cata f (cons i l) = f (Cons i (cata f l)))
cata f (mu x)     = f (Mu (cata c . x . Place))
cata f (Place x)  = x
```

- Map

```
map :: (Int -> Int) -> IList a -> IList a
map f = cata (\x -> case x of
               Cons i tl -> cons (f i) tl
               Mu y -> Mu y)
```

Infinite List Ex

This function is not parametric in x.

- Define the natural numbers as

```
nat = Mu(\x -> Cons(1, map (\y -> y + 1) x))
```

- Define even numbers by mapping agai

Place from outer map consumed by inner map

```
map (\z -> 2*z)
   (Mu(\x -> Cons(1, map (\y -> y +      x))) A
Mu(\x ->
  Cons(2, map (\z -> 2*z)
            (map (\y -> y + 1) (Place x)))) Ã
Mu(\x -> Cons(2, map (\z -> 2*z) x))
```

- This isn't the list of evens, it is the powers of two!

What happened?

- When outer catamorphism introduced a **`Place`**, it was incorrectly consumed by the inner catamorphism.
- The problem is that **`Mu`**’s function isn’t parametric in its argument.
- Using **`Place`** as an inverse can produce incorrect results when the embedded functions are not parametric.

Catamorphisms over non-parametric data

- Is this a problem?
  - Algebraic reasoning only holds for parametric data structures.
  - Can’t tell whether a data structure is well formed from its type.
- Fegarus and Sheard’s solution:
  - Make cata primitive—the user cannot use Place.
  - Tag the type of datastructures that are not parametric.
  - Can’t use cata for those datatypes.

Using Parametricity to Enforce Parametricity

- Our solution: “Tag” parametric datatypes with first-class polymorphism.
- Doesn’t require a special type system -- can be implemented in off-the-shelf languages.
  - Implemented in Haskell.
  - Also possible in OCaml.
- Allows algebraic reasoning.

Intuition

- An expression of type `forall a. Exp a` cannot contain `Place` as that would constrain `a`.

```
lam :: (Exp a -> Exp a) -> Exp a
app :: Exp a -> Exp a -> Exp a

lam (\x -> app (Place int) x) :: Exp Int
```

Iteration over HOAS

- Restrict argument of iteration operator to parametric datatypes

```
 iter :: (ExpF b -> b) ->
              (forall a. Exp a) -> b
```

- In an expression `(lam (\x -> …))` can't iterate over **`x`** because it doesn't have the right type.

```
lam :: (Exp a -> Exp a) -> Exp a
```

Non-parametric Example

- What if we wanted a non-parametric datatype?

```
cata :: (ExpF a -> a) -> Exp a -> a
countvar :: Exp Int -> Int
```

- Lack of parametricity shows up in its type.

```
badexp :: Exp Int
badexp =
lam (\x ->
    if (countvar x) == 1
    then app x x else x)
```

Open Terms

- We have only discussed representing closed ¸-terms. How do we represent open terms?
- Abstraction is used to encode variable binding in the object language.
- Use the same mechanism for free variables. Term with a free variable is a function.

```
(forall a. Exp a -> Exp a)
```

- We can represent ¸-terms with an arbitrary number of free variables using a list.

```
(forall a. [Exp a] -> Exp a)
```

Iteration for arbitrary type constructors

- Problem: **`iter0`** only operates on closed terms of the ¸-calculus.
- **`iter1`** operates on expressions with one free variable.

```
 iter1 ::
   (ExpF b -> b) ->
   (forall a. Exp a -> Exp a) ->
   (b -> b)
```

An Example with Open Terms

```
freevarused ::
 (forall a. Exp a -> Exp a) -> Bool
freevarused e =
 (iter1 (\x ->
          case x of
           (App x y ) -> x || y
           (Lam f) -> f False))
    e
    True
```

Generalizing Iteration Further

- Why not iterate over a list of expressions too?

```
iterList ::(ExpF b -> b) ->
                (forall a. [Exp a]) -> [b]
```

- There are an infinite number of iteration functions we might want.
- Define a single function by abstracting over the type constructor **`g`**.

```
iter ::(ExpF b -> b) ->
       (forall a. g (Exp a)) -> g b
```

- No analogue in Fegarus and Sheard's system.

Implementation of iter

- Can implement all datatypes and iteration operators and in System F
  - Variant of Church encoding.
  - Don’t need explicit recursive type.
  - This implementation has several nice properties.

Properties of Iteration

- Iteration is strongly normalizing.
  - Arg to iter must also be expressible in System F.
- Fusion Law, follows from free theorem:
  - If f, f' are strict functions such that

```
f . f′ = id
```

and

```
f . g = h . bimap(f,f′)
```

Map for datatypes with embedded functions

  - Then

```
f . iter0 g = iter0 h.
```

Connection with Previous Work

- How does this solution to the calculus of Schürmann, Despeyroux, and Pfenning ?
- The SDP calculus:
  - Enforces parametricity using modal types.
  - Was developed for use in logical frameworks.
  - Was the inspiration for our generalized iteration operator.

Modal Types

- Boxed types (□¿) correspond to modal necessity in logic via the Curry-Howard Isomorphism.
  - Propositions are necessarily true if they are true in all possible worlds.
- Used in typed languages to:
  - Describe terms that contain no free variables.
  - Express staging properties of expressions.
  - Enforce parametricity of functions.

Modal Types

- Two contexts, ¢ and ¡, for assumptions that are available in all worlds and those in the present world.
- Introduction

$$\frac{\text{¢ ; ` M : ¿}}{\text{¢;¡ ` }\textbf{box}\text{ M : }\Box\text{¿}}$$

- Elimination

$$\frac{\text{¢;¡ ` M}_1\text{ : }\Box\text{¿}_1 \qquad \text{¢, x: ¿}_1\text{;¡ ` M}_2\text{ : ¿}_2}{\text{¢;¡ ` }\textbf{let box}\text{ x = e}_1\textbf{ in }\text{e}_2\text{ : ¿}_2}$$

Modal Parametricity

- SDP enforces parametricity by distinguishing between “pure” and “impure types”.
- Pure types are those that do not contain boxed types.
  - Exp is a type constant like int (and therefore pure).
  - Term constants for data constructors

    app : Exp ′ Exp → Exp, **lam** : (Exp → Exp) → Exp
- Only allow iteration over terms of *boxed pure* type. □Exp, □(Exp → Exp), etc.

Enforcing Parametricity

- ¸-abstractions have the form:

  lam (¸x:Exp.

  ….

)
- Because x does not have a boxed type, it cannot be analyzed.
- Cannot convert x to a boxed type because it will not be in scope inside of a **box** expression.

Example in SDP

countvar = ¸x:□Exp.

iter[int][**app**)

¸x:int£int. (**fst** x) + (**snd** x),

lam)

¸f:int ! int. f 1] x

Connection with Our Work

- We can encode the SDP calculus into System F using our iteration operator.
  - Very close connection: SDP **iter** translates to our generalized **`iter`**.
- Intuition:
  - Uses universal quantification to explain modality, as in Kripke semantics.
  - Term translation parameterized by the “current world”.
  - Terms in Δ are polymorphic over all worlds. Must be instantiated with current world when used.
  - i.e. encode □Exp as **`(forall a. Exp a)`**

Properties of the Encoding

- Static correctness
  - If a term is well-typed in the SDP calculus, its encoding into System F is also well-typed.
- Dynamic correctness
  - If M evaluates to V in SDP and M translates to e and V translates to e', then e is ¯´-equivalent to e'.

Future Work -- Case Analysis

- There are some functions over datatypes that cannot be written using catamorphisms.
  - Testing that an expression is a ¯-redex.
- SDP introduces a distinct case operator.
  - Theory is complicated.
  - Not obvious whether it can be encoded as we did for iteration.
- Fegarus and Sheard also have a limited form of case.

Future Work -- `coiter`

- Consider the dual to iteration that produces terms with diamond type (modal possibility).

```
data Dia a = Roll (ExpF (Dia a), a)
coiter0 :: (a -> f a)
                 -> a -> (exists a. Dia a)
```

  - Existentials correspond to diamonds (exists a world).
- Is coiteration analogous to anamorphism as iteration is to catamorphism?
- Not obvious how to use coiter
  - Elimination form for possibility only allows use in another term with a diamond type.
  - If we could use iteration on the result it would allow for general recursion.

Conclusions

- Datatypes with embedded functions are useful.
  - Killer app: HOAS
- Easier to iterate over parametric datatypes.
- Do not need tagging or modal necessity for to enforce parametricity -- first-class polymorphism is sufficient.
- Can be implemented entirely in System F.
- Provides an interpretation of modal types.

Implementation in Haskell

- Encode datatypes using a variation on standard trick for covariant datatypes in System F. Encode as an elimination form.

```
type Exp a = (ExpF a -> a) -> a
```

- Generalize our interface from **`ExpF`** to arbitrary type constructors **`f`**.

```
type Rec f a = (f a -> a) -> a
type Exp a = Rec ExpF a
```

Implementation in Haskell

- Encoding datatypes as as elimination forms.
- Implement **`roll`** so that given an elimination function, it invokes iteration.

```
roll :: f (Rec f a) -> Rec f a
roll x = \y -> y (openiter y x)
```

- Here **`openiter`** maps iteration over **`x`**.

```
openiter :: (f a -> a)
            -> g (Rec f a) -> g a
```

- How do we implement **`openiter`**?

Implementation in Haskell

- Because we defined datatypes as their elimination form, basic iteration is just function application.

```
openiter0 :: (f a -> a) -> Rec f a -> a
openiter0 x y = y x
```

- The most general type assigned by Haskell doesn't enforce parametricity, so annotation is needed.

```
iter0 ::
  (f a -> a) -> (forall b. Rec f b) -> a
iter0 = openiter0
```

- Still need to generalize to arbitrary datatypes.

Implementation in Haskell

- To implement the most general form of **`iter`**, we need a mechanism to map over datatypes.
- We can define this function using a polytypic programming. In Generic Haskell:

```
xmap{| f :: * -> * |} ::
  (a -> b, b -> a) ->
  (f a -> f b, f b -> f a)
```

- **`xmap`** generalizes **`map`** to datatypes with positive *and negative* occurrences of the recursive variable.
- Just syntactic sugar, we could implement this directly in Haskell.

Example Instantiation of **`xmap`**

- Expansion of **`xmap{|ExpF|}`** :

```
xmapExpF ::
    (a -> b, b -> a) ->
    (ExpF a -> ExpF b, ExpF b -> ExpF a)
xmapExpF (f,g) (App t1 t2) =
 (App (f t1) (f t2), App (g t1) (g t2))
xmapExpF (f,g) (Lam t)  =
 (Lam (f . t . g), Lam (g . t . f))
```

Implementation in Haskell

- Lift **`openiter0`** to all regular datatypes using **`xmap`**:

```
 openiter{| g : * -> * |} ::
   (f a -> a) -> g (Rec f a) -> a
openiter{| g : * -> * |} x =
  fst (xmap{|g|} (openiter0 x, place))
```

- But we need an inverse to **`openiter0`** for **`xmap`**. Terms are parametric, so we can use the place trick.

```
place :: a -> Rec f a
place x = \y -> x
```

Implementation in Haskell

- Finally, **`iter`** is just **`openiter`** with the appropriate type annotation:

```
iter{| g : * -> * |} ::
    (f a -> a) ->
    (forall b. g (Rec f b))-> g a
iter{| g : * -> * |} = openiter{|g|}
```

Pretty-Printing with Place

- Pretty-printing expressions

```
vars = [ i ++ show j | i <- [ "a" .. "z" ] |
         j <- [1..] ]
showexp :: Exp String -> String
showexp e =
(cata
  (\x y -> \vars ->
    "(" ++ (x vars) ++ " " ++ (y vars) ++ ")")
  (\f  -> \(v:v') ->
     "(\ " ++ v ++ "." ++
          (f (\vars -> v) v') ++ ")")
    e) vars
```

HOAS Interface in Haskell

- Concentrate on the interface for now.

```
data ExpF a = Lam (a -> a)
            | App a a
type Exp a
roll :: ExpF (Exp a) -> Exp a
```

- **`Exp`** is the fix-point of **`ExpF`**.
- Use **`roll`** to coerce into **`Exp`**.

HOAS in Haskell

- Provide helpers to hide **`roll`**.

```
lam :: (Exp a -> Exp a)-> Exp a
lam x = roll (Lam x)
app :: Exp a -> Exp a -> Exp a
app x y = roll (App x y)
```

- How do we iterate over an HOAS expression implemented as **`Exp`**?

Broken Example Continued

- What happens if we try to use **`baditer0`** on **`badexp`**?

  ```
  baditer0 countvar_aux badexp
  ```

- Get 2? Does this make sense? **`badexp`** actually contains four variables.
- Can't pretty-print **`badexp`**, would need type **`Exp String`**.

Broken Example Continued

- Doesn't actually correspond to a term in ¸-calculus.
- **`badexp`** makes assumptions about its type argument forcing it to be **`Exp Int`** instead of **`Exp a`**.
- Problem doesn't exist with **`iter0`** because it enforces parametricity.
- If we used iter0 the previous example wouldn't type check.

Overview of Encoding SDP

- Parameterize the encoding by a “world”, implemented as a type.
- As for our Haskell implementation, encode datatypes as their elimination form.
  - b l$_{¿}$ (§* ¿ ! ¿) ! ¿ encoding of the base type.
  - §* encoding of a signature, ¿ the present world.
- Use type abstraction to enforce parametricity.
  - If ¿$_{1}$ l$_{®}$ ¿$_{2}$ then □¿$_{1}$ l$_{¿}$ 8®.¿$_{2}$
  - Boxed terms can be viewed as functions from an arbitrary world to a well-typed term.

Encoding SDP Terms

- Return to our running example.

  § = **app** : b £ b ! b, **lam** : (b ! b) ! b
- Signature encoded as variant type constructor:

  §*= ¸®.h**app** : ® £ ®, **lam** : ® ! ®i
- Encoding the constructors:
  - **app** B$_¿$ ¸x: ((§* ¿ ! ¿) ! ¿)£((§* ¿ ! ¿) ! ¿).

    roll(inj$_{\textbf{app}}$x **of** §* ¿)
  - **lam** B ¿ ¸x: ((§* ¿ ! ¿) ! ¿) !((§* ¿ ! ¿) ! ¿).

    roll(inj$_{\textbf{lam}}$x **of** §* ¿)

Encoding SDP Terms

- Encoding a use of iteration:

  (**countvar** = ¸x:□b.
  iter[int][**app**)¸x:int£int. (**fst** x) + (**snd** x),
  lam)¸f:int ! int. f 1] x) B ¿
  (**countvar** = ¸x: 8®.((§* ® ! ®) ! ®).
  iter{|¸®. ®|}[int] (¸y:§* int. **case** y
  of inj$_{\mathbf{app}}$ u) (¸x:int£int. (**fst** x) + (**snd** x)) u
  | **inj**$_{\mathbf{lam}}$ v) (¸f:int ! int. f 1) v) x)