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

Stephanie Weirich
University of Pennsylvania

Joint work with Geoff Washburn
Catamorphisms

- Catamorphisms (bananas -- \((\ ))\) are “folds” over data structures.
  - `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 \(\lambda\)-calculus in Haskell
  
  \[
  \text{data Exp} = \text{Var} \ 	ext{String} \\
  \quad \mid \text{Lam} \ 	ext{String} \ 	ext{Exp} \\
  \quad \mid \text{App} \ 	ext{Exp} \ 	ext{Exp}
  \]

• 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.

\[
\text{data Exp = Lam (Exp -> Exp) | App Exp Exp}
\]

- 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
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:

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

• List of ones
  
  ```haskell
  ones = mu (\x -> cons 1 x)
  ```

• Alternating 1’s and 0’s
  
  ```haskell
  onezero = mu (\x -> cons 1 (cons 0 x))
  ```
Using Infinite Lists

• Catamorphism

\[
\text{cata} :: (\text{IListF} \ a \rightarrow a) \rightarrow \text{IList} \ a \rightarrow a
\]
\[
\text{cata} \ f \ (\text{cons} \ i \ l) = f \ (\text{Cons} \ i \ (\text{cata} \ f \ l))
\]
\[
\text{cata} \ f \ (\mu x) = f \ (\text{Mu} \ (\text{cata} \ c \ . \ x \ . \ \text{Place}))
\]
\[
\text{cata} \ f \ (\text{Place} \ x) = x
\]

• Map

\[
\text{map} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{IList} \ a \rightarrow \text{IList} \ a
\]
\[
\text{map} \ f = \text{cata} \ (\mu x \rightarrow \text{case} \ x \ \text{of} \\
\quad \text{Cons} \ i \ tl \rightarrow \text{cons} \ (f \ i) \ tl \\
\quad \text{Mu} \ y \rightarrow \text{Mu} \ y)
\]
Infinite List Example

- Define the natural numbers as
  \[ \text{nat} = \text{Mu}(\lambda x \rightarrow \text{Cons}(1, \text{map}(\lambda y \rightarrow y + 1) x)) \]

- Define even numbers by mapping again?
  \[ \text{map}(\lambda z \rightarrow 2 \times z)(\text{Mu}(\lambda x \rightarrow \text{Cons}(1, \text{map}(\lambda y \rightarrow y + 1) x))) \]
  \[ = \text{Mu}(\lambda x \rightarrow \text{Cons}(2, \text{map}(\lambda z \rightarrow 2 \times z)(\text{map}(\lambda y \rightarrow y + 1) (\text{Place} x)))) \]

- This isn’t the list of evens, it is the powers of two!
What happened?

• When outer catamorphism introduced a \texttt{Place}, it was incorrectly consumed by the inner catamorphism.

• The problem is that \texttt{Mu}’s function isn’t parametric in its argument.

• Using \texttt{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.
• 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
  \[\text{iter} :: (\text{ExpF} \ b \rightarrow b) \rightarrow (\forall a. \text{Exp} \ a) \rightarrow b\]

- In an expression (\text{lam} (\lambda x \rightarrow ...)) can’t iterate over \textbf{x} because it doesn’t have the right type.
  \[\text{lam} :: (\text{Exp} \ a \rightarrow \text{Exp} \ a) \rightarrow \text{Exp} \ a\]
Non-parametric Example

• What if we wanted a non-parametric datatype?
  \[\text{cata} :: (\text{ExpF } a \to a) \to \text{Exp } a \to a\]
  \[\text{countvar} :: \text{Exp Int} \to \text{Int}\]

• Lack of parametricity shows up in its type.
  \[\text{badexp} :: \text{Exp Int}\]
  \[\text{badexp} = \lambda x \to\]
  \[\text{if } (\text{countvar } x) == 1 \text{ then app } x \times x \text{ else } x\]
Open Terms

• We have only discussed representing closed \(\lambda\) - 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.
  \[(\forall a. \text{Exp} \ a \rightarrow \text{Exp} \ a)\]
• We can represent \(\lambda\)-terms with an arbitrary number of free variables using a list.
  \[(\forall a. \left[\text{Exp} \ a\right] \rightarrow \text{Exp} \ a)\]
Iteration for arbitrary type constructors

• Problem: \texttt{iter0} only operates on closed terms of the \(\lambda\)-calculus.

• \texttt{iter1} operates on expressions with one free variable.

\[
\texttt{iter1} :: \\
(\text{ExpF}\ b \rightarrow b) \rightarrow \\
(\text{forall}\ a.\ \text{Exp}\ a \rightarrow \text{Exp}\ a) \rightarrow \\
(b \rightarrow b)
\]
An Example with Open Terms

\[\text{freevarused} :: (\forall a. \text{Exp} \ a \rightarrow \text{Exp} \ a) \rightarrow \text{Bool}\]

\[
\text{freevarused} \ e = (\text{iter1} (\lambda x \rightarrow \\
\text{case } x \text{ of} \\
\quad (\text{App } x \ y) \rightarrow x \lor y \\
\quad (\text{Lam } f) \rightarrow f \ False))
\]

\[e \rightarrow \text{True}\]
Generalizing Iteration Further

• Why not iterate over a list of expressions too?
  \texttt{iterList} :: (\texttt{ExpF} \ b \rightarrow \ b) \rightarrow
  (\forall \ a. \ [\texttt{Exp} \ a]) \rightarrow \ [\ b]

• There are an infinite number of iteration functions we might want.

• Define a single function by abstracting over the type constructor \texttt{g}.
  \texttt{iter} :: (\texttt{ExpF} \ b \rightarrow \ b) \rightarrow
  (\forall \ a. \ g \ (\texttt{Exp} \ a)) \rightarrow \ 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
    \[ f \cdot f' = \text{id} \]
    and
    \[ f \cdot g = h \cdot \text{bimap}(f,f') \]
  – Then
    \[ f \cdot \text{iter0} \ g = \text{iter0} \ h. \]
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 ($\Box \cdot$) 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, $\mathcal{C}$ and $\mathcal{i}$, for assumptions that are available in all worlds and those in the present world.

• Introduction

\[
\begin{align*}
\mathcal{C} ; \ \ M : \mathcal{I} \\
\mathcal{C} ; i \ \ \ \ \text{box} \ M : \square \mathcal{I}
\end{align*}
\]

• Elimination

\[
\begin{align*}
\mathcal{C} ; i \ \ M_1 : \square \mathcal{I}_1 & \quad \mathcal{C} , \ x : \mathcal{I} ; i \ \ M_2 : \mathcal{I}_2 \\
\mathcal{C} ; i \ \ \text{let box} \ x = e_1 \ \ \text{in} \ e_2 : \mathcal{I}_2
\end{align*}
\]
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
    \[\text{app} : \text{Exp ' Exp} \rightarrow \text{Exp}, \text{lam} : (\text{Exp} \rightarrow \text{Exp}) \rightarrow \text{Exp}\]
• Only allow iteration over terms of \textit{boxed pure} type. \[\square \text{Exp}, \square(\text{Exp} \rightarrow \text{Exp}), \text{etc.}\]
Enforcing Parametricity

• ω-abstractions have the form:
  \[
  \text{lam } (\omega x: \text{Exp.} \quad \text{....} \quad )
  \]

• 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 = \( \mu x: \square \text{Exp.} \)

\[ \text{iter}[\text{int}][\text{app} \ ) \]

\( \mu x: \text{int} \text{\int. } (\text{fst} \ x) + (\text{snd} \ x), \)

\[ \text{lam} \ ) \]

\( \mu f: \text{int} \text{\int. } f 1 \ ) \ x \)
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 $\equiv'$-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 $\sim$-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).
  
  \[
  \text{data Dia } a = \text{Roll} \ (\text{ExpF} \ (\text{Dia} \ a), \ a) \\
  \text{coiter0} :: (a \to f \ a) \\
  \quad \to a \to (\exists a. \ Dia \ a)
  \]

  – 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.

\[
\text{type } \text{Exp} \ a = (\text{ExpF} \ a \to a) \to a
\]

• Generalize our interface from \(\text{ExpF}\) to arbitrary type constructors \(\text{f}\).

\[
\text{type } \text{Rec} \ f \ a = (f \ a \to a) \to a
\]

\[
\text{type } \text{Exp} \ a = \text{Rec} \ \text{ExpF} \ a
\]
Implementation in Haskell

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

\[
\text{roll} :: f \ (\text{Rec} \ f \ a) \rightarrow \text{Rec} \ f \ a
\]

\[
\text{roll} \ x = \lambda y \rightarrow y \ (\text{openiter} \ y \ x)
\]

• Here openiter maps iteration over \( x \).

\[
\text{openiter} :: (f \ a \rightarrow a) \\
\rightarrow g \ (\text{Rec} \ f \ a) \rightarrow g \ a
\]

• 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:
  
  ```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 \texttt{xmap}

- Expansion of \texttt{xmap\{ |ExpF| \}}:

\[
\text{xmapExpF} :: (a \rightarrow b, b \rightarrow a) \rightarrow (\text{ExpF} a \rightarrow \text{ExpF} b, \text{ExpF} b \rightarrow \text{ExpF} a) \\
\text{xmapExpF} (f,g) (\text{App } t1 t2) = (\text{App } (f t1) (f t2), \text{App } (g t1) (g t2)) \\
\text{xmapExpF} (f,g) (\text{Lam } t) = (\text{Lam } (f \cdot t \cdot g), \text{Lam } (g \cdot t \cdot f))
\]
Implementation in Haskell

• Lift \texttt{openiter0} to all regular datatypes using \texttt{xmap}:

\[
\begin{align*}
\text{openiter}\{|\ g : * \to * |\} & : : \\
(f \ a \to \ a) & \to g \ (\text{Rec } f \ a) \to a \\
\text{openiter}\{|\ g : * \to * |\} \ x & = \\
\text{fst} \ (\text{xmap}\{|g|\} \ (\text{openiter0} \ x, \text{place}))
\end{align*}
\]

• But we need an inverse to \texttt{openiter0} for \texttt{xmap}. Terms are parametric, so we can use the place trick.

\[
\begin{align*}
\text{place} & : : a \to \text{Rec } f \ a \\
\text{place} \ x & = \ \backslash y \to x
\end{align*}
\]
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

```haskell
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`?
  
  \[ \text{baditer0 countvar_aux badexp} \]

- 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 \( \lambda \)-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.
  – $b \mid_{\varnothing} (\$^* \varnothing ! \varnothing) ! \varnothing$ encoding of the base type.
  – $\$^*$ encoding of a signature, $\varnothing$ the present world.
• Use type abstraction to enforce parametricity.
  – If $\varnothing_1 \mid_{\circ} \varnothing_2$ then $\Box \varnothing_1 \mid_{\varnothing} 8^\circ.\varnothing_2$
  – Boxed terms can be viewed as functions from an arbitrary world to a well-typed term.
Encoding SDP Terms

• Return to our running example.
  \[ \& = \text{app} : b \ & b \ ! \ b, \ \text{lam} : (b \ ! \ b) \ ! \ b \]

• Signature encoded as variant type constructor:
  \[ \&^* = \varnothing.\text{app} : \varnothing \ & \ \varnothing, \ \text{lam} : \varnothing \ ! \ \varnothing \]

• Encoding the constructors:
  - \[ \text{app} \ B_\varnothing \ , \ x : (((\&^* \ ? \ ! \ ?) \ ! \ ?) \ & \ ((\&^* \ ? \ ! \ ?) \ ! \ ?). \]
    \[ \text{roll(inj}_{\text{app}x \ of \ \&^* \ ?) \]
  - \[ \text{lam} \ B_\varnothing \ , \ x : (((\&^* \ ? \ ! \ ?) \ ! \ ?) ! ((\&^* \ ? \ ! \ ?) \ ! \ ?). \]
    \[ \text{roll(inj}_{\text{lam}x \ of \ \&^* \ ?) \]

8/24/06
Encoding SDP Terms

• Encoding a use of iteration:

\[
\begin{align*}
\text{(countvar } &= \; \{ x: \Box b. \\
&\text{ iter[int][ } app \}; x: \text{int}\geq\text{int. (fst } x) + (\text{snd } x), \\
&\text{ lam } \}; f:\text{int ! int. f 1 ] x} \} \text{ B } \\
\text{(countvar } &= \; \{ x: 8\Box.((\Box^* \Box ! \Box)! \Box ).\\
&\text{ iter{ }l\Box. \Box}|\text{][int] } \{ y:\Box^* \text{ int. case } y \\
&\text{ of inj_{app } u } \}) \{ x: \text{int}\geq\text{int. (fst } x) + (\text{snd } x)) u \\
&| \text{ inj_{lam } v } \}) \{ f:\text{int ! int. f 1) v } \} \text{ x)}
\end{align*}
\]