Recursion Parameterised by Monads:

Characterisation and Examples

Johan Glimming

glimming@kth.se

Stockholm University (KTH)

Sweden

9 December 2003

Aims

- Describe the extension of folds to monadic folds, and present state monads via example.

Aims

- Describe the extension of folds to monadic folds, and present state monads via example.
- Explain the required lifting from F-algebras to F^M-algebras [Beck 1969, Fokkinga 1994, Pardo 2001].

Aims

- Describe the extension of folds to monadic folds, and present state monads via example.
- Explain the required lifting from F-algebras to F^M-algebras [Beck 1969, Fokkinga 1994, Pardo 2001].
- Conclude by (briefly) describing on-going research on the semantics of objects with method update [Glimming, Ghani 2003].

Folds, Monads, and Monadic Folds

Recursion = fold

- Structural recursion is captured by a combinator called `fold`, i.e. $\texttt{fold}_{List}$, $\texttt{fold}_{Nat}$, $\texttt{fold}_{Tree}$, ...

```
sum [] = 0
sum x:xs = x + sum xs
```

- Let *List* denote lists of natural numbers, e.g. `[881,883,887]`.

Recursion = fold

- The important part of this schema is $0 : \tau$ and $+ : \tau \times \tau \rightarrow \tau$ where τ is *Nat* for `sum`.

Recursion = fold

- The important part of this schema is $0 : \tau$ and $+ : \tau \times \tau \to \tau$ where τ is *Nat* for $\mathtt{sum}$.
- $(\textit{Nat}, [0, +])$ forms an algebra of the same pattern functor as $(\textit{List}, [[], :])$.

Recursion = fold

- The important part of this schema is $0 : \tau$ and $+ : \tau \times \tau \to \tau$ where τ is *Nat* for `sum`.
- $(\mathit{Nat}, [0, +])$ forms an algebra of the same pattern functor as $(\mathit{List}, [[], :])$.
- $\texttt{fold}_{\mathit{List}}$ has type $\tau \times (\tau \times \tau \to \tau) \times \mathit{List} \to \tau$.

Recursion = fold

- The important part of this schema is $0 : \tau$ and $+ : \tau \times \tau \to \tau$ where τ is *Nat* for `sum`.
- $(\mathit{Nat}, [0, +])$ forms an algebra of the same pattern functor as $(\mathit{List}, [[], :])$.
- $\mathtt{fold}_{\mathit{List}}$ has type $\tau \times (\tau \times \tau \to \tau) \times \mathit{List} \to \tau$.
- Write $(\![f]\!)_F$ for fold/catamorphism. f can be $[0, +], [[x], :], \ldots$

Why bother?

- Language constructs (c.f. Charity) – increasing the **expressive** power.

Why bother?

- Language constructs (c.f. Charity) – increasing the **expressive** power.
- **Structure** programs after the type of values functions consumes.

Why bother?

- Language constructs (c.f. Charity) – increasing the **expressive** power.
- **Structure** programs after the type of values functions consumes.
- **Calculation**: we get nice *theorems for free* and we can use these theorems to develop programs in a style similar to the way an engineer works with calculus when he builds a bridge.

Why bother?

- Language constructs (c.f. Charity) – increasing the **expressive** power.
- **Structure** programs after the type of values functions consumes.
- **Calculation**: we get nice *theorems for free* and we can use these theorems to develop programs in a style similar to the way an engineer works with calculus when he builds a bridge.
- **Optimisation**: acid rain, fusion (in compiler?)

Recipe for a monad: state

- Start with your favourite type constructor and a pinch of functoriality:

```
newtype State s a =
  State {runState :: s -> (a,s)}
```

Cont'd

- Put it on an ad-hoc plate, use all strength:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

instance Monad (State s) where
  return a = State $ \s ->
    (a, s)
  m >>= k  = State $ \s ->
    let (a, s') = runState m s
    in
    runState (k a) s'
```

Computational intuition

- **State monad:** the monad itself is a mappping from an initial state to a value and a new state. The two operations (unit and bind) corresponds to:

1. making a value a state-based computation, $a \mapsto \lambda s.(a, s)$, and
2. $f >>= g$ means forming a new computation that, given an initial state s, evaluates f in that state, moving to a new state, in which g is evaluated: $s \mapsto g\, a\, s'$ where $(a, s') = f\, s$.

Cont'd recipe ...

- Serve hot with a fold for the season (this year, it's lists):

```
foldrM :: (Monad m) =>
  (a -> b -> m a) -> a -> [b] -> m a
foldrM _ a []       =  return a
foldrM f a (x:xs) =
  f a x >>= \fax -> foldM f fax xs
```

- Now you've baked a state, and you can fold it!

Kleisli triples

- Let's make sure:

1. Left unit:
   `return a >>= k = k a`
2. Right unit:
   `k >>= result = k`
3. Associativity of bind:
   `(a >>= b) >>= c = a >>= (b >>= c)`

Kleisli triples – or triples?

- In fact, this is a *Kleisli triple* (an object construction) rather than a monad, but Kleisli triples are in bijective correspondence with monads (a k a triples).

Example: accumulate

We now consider a function that, given a list of computations of the form $M\ A$, produces a computation that collects the result of all those computations, from left to right:

```
accumulate :: Monad m => [m a] -> m [a]
accumulate = foldr (\ ma mas ->
                      do a  <- ma
                         as <- mas
                         (a:as))
                   (return [])
```

Cont'd

This `accumulate` function can be written as a monadic (left) fold on lists:

```
accumulate = foldlM (\ as ma ->
                      do a <- ma
                         (as ++ [a]))
                    (return [])
```

From this one may (correctly) guess that `foldlM` can be written in terms of `foldl`.

Why bother?

- **Structure** programs after the sort of computation of value they produce [Meijer and Jeuring 1995].
- **Examples:** exceptions, layers of state-based computation, non-determinism, partiality, ...
- **For our purposes:** monads gives rise to a *higher-order* denotation of objects where state is captured by a state transforming function, and an object becomes a functional. But *is monadic fold general enough?*

Entering the world of semantics ...

- Types = objects in suitable category ($\mathbf{Set}$, $\mathbf{Cpo}_\perp$), and computations = arrows.

Entering the world of semantics ...

- Types = objects in suitable category (**Set**, **Cpo**$_\perp$), and computations = arrows.
- Recursive datatype is represented by a *pattern functor*, e.g. $\mathsf{F}X = \mathbf{1} + X \times X + \mathbb{N}$. Fixpoint is the datatype (object). Initial algebra is type constructor e.g. $[\mathit{Nil}, \mathit{Cons}]$.

Entering the world of semantics ...

- Types = objects in suitable category ($\mathbf{Set}$, $\mathbf{Cpo}_{\perp}$), and computations = arrows.
- Recursive datatype is represented by a *pattern functor*, e.g. $\mathsf{F}X = \mathbf{1} + X \times X + \mathbb{N}$. Fixpoint is the datatype (object). Initial algebra is type constructor e.g. $[Nil, Cons]$.
- The fixpoint can be identified with the initial algebra over the pattern functor.

Entering the world of semantics ...

- Types = objects in suitable category ($\mathbf{Set}, \mathbf{Cpo}_\perp$), and computations = arrows.
- Recursive datatype is represented by a *pattern functor*, e.g. $\mathsf{F}X = \mathbf{1} + X \times X + \mathbb{N}$. Fixpoint is the datatype (object). Initial algebra is type constructor e.g. $[Nil, Cons]$.
- The fixpoint can be identified with the initial algebra over the pattern functor.
- **Goal:** What is the semantics of *monadic* folds? *Useful for recursion on objects?*

Polynomial functors

The smallest class of functors closed under composition and containing the following basic functors:

- n-ary constant functor $\underline{A}^n$ for fixed A, n:

$$\begin{aligned} \underline{A}^n f_0 \ldots f_{n-1} &= id_A \\ \underline{A}^n B_0 \ldots B_{n-1} &= A \end{aligned}$$

- n-ary projection functors π_i^n:

$$\begin{aligned} \pi_i^n f_0 \ldots f_{n-1} &= f_i \\ \pi_i^n A_0 \ldots A_{n-1} &= A_i \end{aligned}$$

- id, the identity functor
- $+$, the sum functor
- $\times$, the product functor

(Regular functors)

- The class of polynomial functors can be extended to the slightly larger class of *regular functors* by also considering:
- **Type functors** – i.e. fixpoints of parameterised regular functors, e.g. *List* α (which is in fact a bifunctor) *plus* operation on arrows which is usually defined to be the `map` operation.

F-algebras

Given:

- endofunctor $\mathsf{F} : \mathcal{C} \longrightarrow \mathcal{C}$

Define: F-algebra is an arrow $\sigma : \mathsf{F}\, A \longrightarrow A$ in $\mathcal{C}$

- We write $(A, \sigma)_{\mathsf{F}}$
- A is the carrier of the F-algebra
- F determines the signature (or operational type)
- σ is the structure (or operation)

F-homomorphisms

Structure-preserving mapping from the carrier of one F-algebra to the carrier of another F-algebra:

A homomorphism $\phi : X \longrightarrow Y$ from (X, σ) to (Y, τ) is defined by the (universal) property:

$$\boxed{\phi \circ \sigma = \tau \circ \mathsf{F}\,\phi}$$

$$\begin{tikzcd}
\mathsf{F}\,X \arrow[r, "\sigma"] \arrow[d, "F\,\phi"'] & X \arrow[d, "\phi"] \\
\mathsf{F}\,Y \arrow[r, "\tau"'] & Y
\end{tikzcd}$$

Example

- $\mathtt{sum}$ is a *List*-homomorphism: it maps F-algebras to F-algebras for the same pattern functor F.

Homomorphism since:

$$\mathtt{sum} \circ [\mathit{Nil}, \mathit{Cons}] = [0, +] \circ \mathsf{F}\,\mathtt{sum}$$

Recursion

- $(\![\ldots]\!)_F$ is the notation for structural recursion over a datatype (primitive if the datatype happens to be the natural numbers).
- Category theory gives us a characterisation as a universal property of (polynomial) datatypes. Existence is immediate in "rich-enough" categories e.g. **Fun** and **Cpo**.

Catamorphism

By definition, there is a unique homomorphism, $h : inn_F \longrightarrow f$, to every F-algebra (A, f) from the initial F-algebra inn_F.

This homomorphism is denoted $(\![f]\!)$ and called the catamorphism for the algebra (A, f).

For every F-algebra (A, f) we can formally define the catamorphism with a *universal property* (arrow):

$$\boxed{h = (\![f]\!) \ \equiv \ h \circ \alpha = f \circ \mathsf{F}\, h}$$

Catamorphism diagram

Let F be some endofunctor, $f : \mathsf{F}\,A \longrightarrow A$ some algebra, and let $(T, inn_{\mathsf{F}})_{\mathsf{F}}$ be the initial algebra. $(\![f]\!)$ is the unique homomorphism that makes the following diagram commute:

$$\begin{tikzcd}
\mathsf{F}\,T \arrow[r, "inn"] \arrow[d, "\mathsf{F}\,(\![h]\!)"'] & T \arrow[d, "(\![h]\!)"] \\
\mathsf{F}\,A \arrow[r, "f"'] & A
\end{tikzcd}$$

Anamorphism

Dually, there is a unique homomorphism, $h : \mathsf{F} \longrightarrow out_{\mathsf{F}}$, to every F-coalgebra (A, f) from the initial F-coalgebra inn_{F}.

This homomorphism is denoted $]\!)f(\![$ and called the anamorphism for the algebra (A, f).

For then F-algebra (A, f) we can formally define the anamorphism with a universal property:

$$\boxed{h =]\!)f(\![\equiv h \circ \alpha = f \circ \mathsf{F}\, h}$$

Lifting Construction

Milestones

- [Beck 1969]: foundational work on algebras over monads and distributive laws.
- [Fokkinga 1994]: provided a lifing construction valid for all regular functors, but not allowing state monads.
- [Pardo 1998, 2001]: worked on the characterisation of monadic lifting of algebras, e.g. by showing that every strong commutative functor can be lifted to a monadic functor using the strength for distribution over product functor.

From $\mathcal{C}$ to $\mathcal{C}^M$ ... and back

Barr and Wells provide us with the following adjunction (pair of functors):

$$\begin{tikzcd} \mathcal{C}^M \arrow[d, shift left=2ex, "U"] \arrow[d, phantom, "\dashv"] \\ \mathcal{C} \arrow[u, shift left=2ex, "\_^M"] \end{tikzcd}$$

Adjunction

... where (remembering the monad)

$$\begin{aligned} \_^{M} &:: \mathcal{C} \longrightarrow \mathcal{C}^{M} \\ A^{M} &= A \\ f^{M} &= \eta \bullet f \end{aligned}$$

and (forgetting it again)

$$\begin{aligned} \mathsf{U} &:: \mathcal{C}^{M} \longrightarrow \mathcal{C} \\ \mathsf{U}\,A &= M\,A \\ \mathsf{U}\,f &= \mu \bullet M\,f \end{aligned}$$

Preservation of limits

Now we have the very important property:

- Left adjoints preserve colimits, e.g. initiality.

Above, $\_^M$ is the left adjoint, and hence an initial object A under $\_^M$, A^M, is also initial in $\mathcal{C}^M$.

Preservation of limits

Now we have the very important property:

- Left adjoints preserve colimits, e.g. initiality.

Above, $\_^M$ is the left adjoint, and hence an initial object A under $\_^M$, A^M, is also initial in $\mathcal{C}^M$.

- **But**... we want to lift objects in the category of F-algebras to the category of F-algebras over a monad M, with the same preservation of initiality.

$Alg(\mathsf{F})$ to $Alg^M(\mathsf{F}^M)$

- What we are looking for:

$$\begin{tikzcd}
Alg^M(\mathsf{F}^M) \arrow[d, shift left=3, "V"] \\
Alg(\mathsf{F}) \arrow[u, shift left=3, "_^M"] \arrow[u, phantom, "\dashv"]
\end{tikzcd}$$

Interlude: F to F^M

- Lifting F-algebras require us first to be able to lift functors alone, i.e. from $\mathsf{F} :: \mathcal{C} \longrightarrow \mathcal{C}$ we want to construct $\mathsf{F}^M :: \mathcal{C}^M \longrightarrow C^M$.

$$\begin{aligned} F^M\, A &= \mathsf{F}\, A \\ F^M\, f &:: \mathsf{F}\, A \longrightarrow M\, (\mathsf{F}\, B) \end{aligned}$$

Interlude: F to F^M

- Lifting F-algebras require us first to be able to lift functors alone, i.e. from $\mathsf{F} :: \mathcal{C} \longrightarrow \mathcal{C}$ we want to construct $\mathsf{F}^M :: \mathcal{C}^M \longrightarrow C^M$.

$$\begin{aligned} F^M\, A &= \mathsf{F}\, A \\ F^M\, f &:: \mathsf{F}\, A \longrightarrow M\, (\mathsf{F}\, B) \end{aligned}$$

But ... setting $\mathsf{F}^M\, f = \mathsf{F}\, f$ would give something of type $\mathsf{F}\, f :: \mathsf{F}\, A \longrightarrow \mathsf{F}\, M\, B$ since $f : A \longrightarrow M\, B$ are the arrows in $\mathcal{C}^M$. How can we get a functor that gives us $M\, (\mathsf{F}\, B)$ as target for such an arrow?

Distribution laws

Consider a family of natural transformations $\delta_{\mathsf{F}} : \mathsf{F}\, M \dot{\longrightarrow} M\, \mathsf{F}$.

- Such δ are called distribution laws because they perform a distribution of a monad over a functor.

Strength – the missing piece

- Fokkinga (1994) gives an definition of a possible δ by induction over the structure of regular functors, assuming distribution over product.

The strength of a monad (M, η, μ) is given by a natural transformation $\tau_{A,B} :: A \times M\,B \longrightarrow M\,(A \times B)$. State monad is strong in both **Sets** and **Cpo**$_\perp$.

- Pardo (2001) demonstrated that every strong monad M has a distribution law for the monad over the product functor, i.e. a natural transformation $\psi_{A,B} :: M\,A \times M\,B \longrightarrow M\,(A \times B)$.

Distribution law for regular F

Following Fokkinga (1994) and Pardo (2001):

$$
\begin{aligned}
\delta^{I}_{A} &= id_M\ A \\
\delta^{\underline{C}}_{A} &= id_C \\
\delta^{\pi^{n_i}}_{(A_1,\ldots,A_n)} &= id_{MA_i} \\
\delta^{\times}_{(A,B)} &= \psi_{(A,B)} \\
\delta^{+}_{(A,B)} &= [M\,\mathsf{inl}, M\,\mathsf{inr}] \\
\delta^{(\mathsf{F}\circ\mathsf{G})} &= \delta^{F}_{(G_i\,A)} \circ \mathsf{F}(\delta^{G_1}_{A}, \ldots, \delta^{G_n}_{A}) \\
\delta^{D}_{A} &= \text{(c.f. Pardo or Fokkinga)}
\end{aligned}
$$

From distributivity to lifting

We now only need to verify that the definition of the lifting given by:

$$\begin{aligned} \mathsf{F}^M\, f &:: \mathsf{F}\, A \longrightarrow M\left(\mathsf{F}\, B\right) \\ \mathsf{F}^M\, f &= \delta^{\mathsf{F}}_B \circ \mathsf{F}\, f \end{aligned}$$

Indeed, the construction has the right type. Fokkinga and Pardo proves that F^M is indeed a functor if F is regular and M is strong (as functor) in the base category.

What about the F-algebras?

We can now use Barr and Wells lifting together with Fokkinga and Pardo's lifting. We have showed *one possible* construction of:

$$\_^{M} :: Alg(\mathsf{F}) \longrightarrow Alg^{M}(\mathsf{F}^{M})$$

Current Research

Pros with monadic folds

- Too specific to be useful ... [Meijer and Jeuring 1995]

Consider `mapl`, a function that maps a monadic function over a list, starting from the left. The definition of this function becomes complicated when written as a monadic fold (c.f. Meijer and Jeuring).

- Pardo (2001), on the other hand, argues that (co)monadic (un)folds “capture functions commonly used in practise” ...

Cons with monadic folds

- State monads are excellent for representing objects [Nordlander 2000].

Cons with monadic folds

- State monads are excellent for representing objects [Nordlander 2000].
- State monads can be used to represent objects *with method update* [Glimming and Ghani].

Cons with monadic folds

- State monads are excellent for representing objects [Nordlander 2000].
- State monads can be used to represent objects *with method update* [Glimming and Ghani].
- **Hence**, perhaps monadic folds can form building blocks for future (real!) O-O programming languages which supports method update. **Q: Why method update?**

Generalisation of the lifting

- Nested datatypes (based on e.g. [Bailey 2000]).

Generalisation of the lifting

- Nested datatypes (based on e.g. [Bailey 2000]).
- Datatypes involving function spaces (*i.e. more than regular!*)

Generalisation of the lifting

- Nested datatypes (based on e.g. [Bailey 2000]).
- Datatypes involving function spaces (*i.e. more than regular!*)
- Variations – establishing a “calculus of monadic liftings” where we can choose from several alternative liftings, each with a clear computational intuition – there are many ways to construct the lifting (for every distributive law...)

Current work - method update

- We seek a denotation of objects that will give us recursion (and corecursion) *for free*.

Current work - method update

- We seek a denotation of objects that will give us recursion (and corecursion) *for free*.
- We deal with objects with *updatable methods* (c.f. Abadi and Cardelli). To our knowledge, such objects are not semantically well-understood to date (c.f. Reus/Stretcher 2001).

Current work - method update

- We seek a denotation of objects that will give us recursion (and corecursion) *for free*.
- We deal with objects with *updatable methods* (c.f. Abadi and Cardelli). To our knowledge, such objects are not semantically well-understood to date (c.f. Reus/Stretcher 2001).
- We currently use the solution to the domain equation: $\mathit{self} \cong \mathit{self} \longrightarrow \mathsf{F}\mathit{self}$ which is equivalent to studying the fixpoint of $\mathsf{G}\,X = (\mathsf{F}\,X)^X$ ... but the occurrence of X is both positive and negative and hence G is *not* a functor.

Current work - method update

- We seek a denotation of objects that will give us recursion (and corecursion) *for free*.
- We deal with objects with *updatable methods* (c.f. Abadi and Cardelli). To our knowledge, such objects are not semantically well-understood to date (c.f. Reus/Stretcher 2001).
- We currently use the solution to the domain equation: $\mathit{self} \cong \mathit{self} \longrightarrow \mathsf{F}\mathit{self}$ which is equivalent to studying the fixpoint of $\mathsf{G}\,X = (\mathsf{F}\,X)^X$ ... but the occurrence of X is both positive and negative and hence G is *not* a functor.
- Freyd provides a technique of transforming this equation into a functor rather than difunctor.