A Haskell Companion for "Fold and Unfold for Program Semantics"

Uk'taad B'mal
The University of Kansas - ITTC
2335 Irving Hill Rd, Lawrence, KS 66045
lambda@ittc.ku.edu

November 22, 2014

Abstract

This document is a primer to accompany the paper "Fold and Unfold for Program Semantics" by Graham Hutton. It attempts to re-implement in `Haskell` the interpreters written in `Gopher`.

1 Introduction

2 Fold For Expressions

Module *FoldForExpressions* presents a Haskell encoding of Hutton's interpreter and definition of fold over expressions:

module *FoldForExpressions* **where**

First, we define data structures for *Expr* representing numbers and addition. This is a parameterized version of the expression data type defined in the Duponcheel interpreters:

$$
\begin{array}{l}
\textbf{data}\ (\mathit{Show}\ a, \mathit{Eq}\ a) \Rightarrow \mathit{Expr}\ a \\
\quad = \mathit{Val}\ a \\
\quad \mid \mathit{Add}\ (\mathit{Expr}\ a)\ (\mathit{Expr}\ a) \\
\quad\ \ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

It's easy now to define direct evaluation of the expression data structure using direct recursion. Note that *Expr* is instantiated over *Int* in this evaluator:

$$
\begin{array}{l}
\mathit{eval}_1 :: \mathit{Expr}\ \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{eval}_1\ (\mathit{Val}\ n) = n \\
\mathit{eval}_1\ (\mathit{Add}\ x\ y) = \mathit{eval}_1\ x + \mathit{eval}_1\ y
\end{array}
$$

The function ϕ is actually defined as *deno* in the Hutton paper. I have used ϕ to be consistent with Duponcheel. The functions f and g provide the semantics for *Val* and *Add*.

$$f = id$$
$$g = (+)$$

$$\phi\ (Val\ n) = f\ n$$
$$\phi\ (Add\ x\ y) = g\ (\phi\ x)\ (\phi\ y)$$

The *fold* operation defines a general fold function over expressions. The signature of the fold is interesting. So much so that I let `Haskell`'s type inference system find it for me. a reflects the type of the value encapsulated by $Val\ a$ while b reflects the domain of what will be the evaluation function. Specifically, $a \to b$ defines the signature of the value extraction function. For $eval_2$, that function is *id* because we are simply extracting the value. For *comp* later, it will be *Inst*, the instruction generated for the stack machine.

$$fold :: (Eq\ a, Show\ a) \Rightarrow (a \to b) \to (b \to b \to b) \to Expr\ a \to b$$
$$fold\ f\ g\ (Val\ n) = f\ n$$
$$fold\ f\ g\ (Add\ x\ y) = g\ (fold\ f\ g\ x)\ (fold\ f\ g\ y)$$

The evaluation function, $eval_1$, defined earlier can now be redefined using *fold* and specifying the semantic mappings for *Val* and *Add* respectively.

$$eval_2 = fold\ id\ (+)$$

Hutton also defines a compiler function, *comp*, that generates a sequence of instructions for a stack machine. *Inst* is a data type representing possible instructions for the machine.

$$\mathbf{data}\ Inst = PUSH\ Int \mid ADD\ \mathbf{deriving}\ (Eq, Show)$$

comp translates an expression defined over integers into a list of instructions. When looking at the *fold*, a is instantiated with *Expr Int* while b is instantiated with $[Inst]$. Initially, I was confused with the definition of g. The fold will actually evaluate the arguments to *Add* and return them. It thus makes sense that g simply concatenates the lists of instructions and tacks an *ADD* onto the end.

$$comp :: Expr\ Int \to [Inst]$$
$$comp = fold\ f\ g$$
$$\quad \mathbf{where}$$
$$\quad\quad f\ n = [PUSH\ n]$$
$$\quad\quad g\ xs\ ys = xs +\!\!+\ ys +\!\!+\ [ADD]$$

Hutton extends $eval_2$ to include variables and a program store. I'm going to extend $eval_2$ like I did for Duponcheel by adding a multiply operation. It's not particularly difficult, however it becomes clear that the interpreter is not particularly modular or extensible.

First, we need to extend the expression data structure to include the multiplication operation:

$$\mathbf{data}\ (Show\ a, Eq\ a) \Rightarrow Expr_1\ a$$
$$= Val1\ a$$
$$\mid Add1\ (Expr_1\ a)\ (Expr_1\ a)$$
$$\mid Mul1\ (Expr_1\ a)\ (Expr_1\ a)$$
$$\mathbf{deriving}\ (Eq, Show)$$

Then we have to redefine *fold* as *fold1* to take three functions to include the semantics for multiply:

$$fold1 :: (Show\ a, Eq\ a) \Rightarrow (a \rightarrow b) \rightarrow (b \rightarrow b \rightarrow b) \rightarrow (b \rightarrow b \rightarrow b) \rightarrow Expr_1\ a \rightarrow b$$
$$fold1\ f\ g\ h\ (Val1\ x) = f\ x$$
$$fold1\ f\ g\ h\ (Add1\ x\ y) = g\ (fold1\ f\ g\ h\ x)\ (fold1\ f\ g\ h\ y)$$
$$fold1\ f\ g\ h\ (Mul1\ x\ y) = h\ (fold1\ f\ g\ h\ x)\ (fold1\ f\ g\ h\ y)$$

Now we can define $eval_3$ to perform the appropriate evaluation:

$$eval_3 = fold1\ id\ (+)\ (*)$$

Hutton continues to generize the evaluation function by adding a *Store* and variable access to the interpetation. He does not provide a full definition for *Store*, so we'll try to here. First we define a new *Expr* type that includes the concept of a variable:

$$
\begin{aligned}
&\mathbf{data}\ Expr_2\ ty \\
&\quad = Val2\ ty \\
&\quad \mid Add2\ (Expr_2\ ty)\ (Expr_2\ ty) \\
&\quad \mid Var2\ String
\end{aligned}
$$

Note that the $Expr_2$ type is parameterized over the contained data, n, and the variable name type, v. This will allow the most general possible *Store* to be defined.

Next, we define a new *fold* that includes the processing of variable references. h now provides a lookup capability for variables in the store.

$$fold2\ f\ g\ h\ (Val2\ x) = f\ x$$
$$fold2\ f\ g\ h\ (Add2\ x\ y) = g\ (fold2\ f\ g\ h\ x)\ (fold2\ f\ g\ h\ y)$$
$$fold2\ f\ g\ h\ (Var2\ c) = h\ c$$

We still need a *Store* and *find* function for records in the store. The easiest *Store* to define is a list of pairs whose first element is a *String* and whose second is a value:

$$\mathbf{data}\ Binding\ n = Binding\ String\ n$$

$$\mathbf{type}\ Store = [(Binding\ Int)]$$

$$find :: String \rightarrow Store \rightarrow Int$$
$$find\ v\ [\,] = error\ (\texttt{"Variable "} +\!\!+\ v\ +\!\!+ \texttt{" not found in store"})$$
$$find\ n1\ ((Binding\ n2\ z) : bs) = \mathbf{if}\ (n1 \equiv n2)\ \mathbf{then}\ z\ \mathbf{else}\ find\ n1\ bs$$

Now we can write the *eval* function over this new data structure with an associated store like Hutton:

$$
\begin{aligned}
&evalS :: (Expr_2\ Int) \rightarrow (Store \rightarrow Int) \\
&evalS = fold2\ f\ g\ h \\
&\qquad\quad \mathbf{where} \\
&\qquad\qquad f\ n = \lambda s \rightarrow n \\
&\qquad\qquad g\ fx\ fy = \lambda s \rightarrow fx\ s + fy\ s \\
&\qquad\qquad h\ n = \lambda s \rightarrow find\ n\ s
\end{aligned}
$$

This function maps an expression to a function mapping *Store* to *Int* rather than just an *Int*. We must in effect provide a *Store* to perform a full evaluation. However, that *Store* is not mutable - there is no mechanism for adding to or deleting from the *Store*.

Some interesting examples:

$$
\begin{aligned}
&\mathit{testS} = (\mathit{Add2}\ (\mathit{Var2}\ \texttt{"Test"})\ (\mathit{Var2}\ \texttt{"x"}))\\
&\mathit{failS} = (\mathit{Add2}\ (\mathit{Var2}\ \texttt{"test"})\ (\mathit{Var2}\ \texttt{"x"}))\\
&\mathit{storeS} :: \mathit{Store} = [(\mathit{Binding}\ \texttt{"Test"}\ 3), (\mathit{Binding}\ \texttt{"x"}\ 4)]
\end{aligned}
$$

You can now enter the following at the command line to see what's going on:

$\mathit{evalS}\ \mathit{storeS}\ \mathit{testS}$

$\mathit{evalS}\ \mathit{storeS}\ \mathit{failS}$

This is interesting from a semantics definition perspective. However, it is not a particularly useful way to define modular interpreters as Duponcheel does.

One question is whether the *Sum* type used buy Duponcheel to form composite data types and composite application functions. Hutton's approach uses the position of the folded function in the argument list to determine which function to apply. Duponcheel defines a function composition operation over the *Sum* type to accomplish a similar task.

First, define new data types for each element of the expression:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Expr}_2\ a = \mathit{Val2}\ a\\
&\mathbf{data}\ \mathit{Expr}_3\ a = \mathit{Add2}\ a\ a
\end{aligned}
$$

Now reproduce the *Sum* definition from Duponcheel:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Sum}\ a\ b\\
&\quad = L\ a\\
&\quad\ \ \ R\ b
\end{aligned}
$$

$$
\begin{aligned}
&(<+>) :: (x \to z) \to (y \to z) \to ((\mathit{Sum}\ x\ y) \to z)\\
&f <+> g = s \to \mathbf{case}\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad (L\ x) \to f\ x\\
&\qquad\qquad\qquad\qquad (R\ x) \to g\ x
\end{aligned}
$$

Now define a *Term* type to represent both possible term elements:

$$\mathbf{type}\ \mathit{Term} = \mathit{Sum}\ (\mathit{Expr}_2\ \mathit{Int})\ (\mathit{Expr}_3\ \mathit{Int})$$

Finally, define a mapping function from the composite of the *id* and (+) operations:

$$j = \mathit{id} <+> (+)$$

If all goes well, we should be able to fold j through the structure.

$$fold2\ j\ (L\ (Val2\ x)) = j\ x$$
$$fold2\ j\ (R\ (Add2\ x\ y)) = j\ (fold2\ j\ x)\ (fold2\ j\ y)$$

But all doesn't go well. The problem is that the arity of j is fixed. Thus, there is no way to apply it to both x in the first case and the results of *fold2* applied twice in the second case. What's happening here is actually pretty simple. Hutton's fold semantics unpackages the arguments to *Add2* before applying *Fold2*. This explains why: (i) the signature for g above maps two elements of the type *encapsulated by Val* to an element of the same type rather than two elements of the *Expr*; and (ii) why the function composition cannot be applied in the same way. It would be interesting to attempt to rewrite the fold to achieve this end.

3 Unfold For Expressions

Module *UnfoldForExpressions* presents a Haskell encoding of Hutton's interpreter and definition of fold over expressions:

```
module UnfoldForExpressions where

data (Show a, Eq a) => Expr a
    = Val a
    | Add (Expr a) (Expr a)
     deriving (Eq, Show)
```

The translation function is interesting as it almost directly implements the operational semantics rules defined for expressions and exhibits why it is an unfold. What *trans* does is takes an expression and generates a list of possible transformations of that expression. Because the semantics of *Add* don't specify whether the first or second argument is evaluated first, there are two possible paths. Thus, *trans* is effectively a one-step unfold.

```
trans :: Expr Int -> [Expr Int]
trans (Val n) = []
trans (Add (Val n) (Val m)) = [Val (n + m)]
trans (Add x y)
      = [Add x' y | x' <- trans x] ++
        [Add x y' | y' <- trans y]
```

Hutton now defines *exec* as a function that applies *trans* repeatedly until nothing remains to be translated. The result is a tree whose nodes are the results of applying *trans* to their immediate predecessor in the tree. So, each list generated by *trans* is treated as the data associated with a set of sub-nodes. To represent this, we'll generate a tree. First, define a data type for a tree with arbitrary subtrees:

```
data Tree a = Node a [Tree a] deriving Show
```

Now *exec* can be defined recursively using list comprehension. Executing e is e itself in a node with the possible translations of e. Because *exec* is referenced recursively in the list comprehension, it will be applied to the results of each application of *trans*. This is a very powerful and interesting application of `Haskell`'s list comprehension operator!

```
exec :: Expr Int -> Tree (Expr Int)
exec e = Node e [exec e' | e' <- trans e]
```

Hutton abstracts from the case for *exec* by defining a function, *oper*, that defines the operational semantics for a tree structure using arbitrary functions for *id* and *trans* as they appear in *exec*. Note that *id* does not explicitly appear, but is implicitly called on the input argument. *oper* as defined in Hutton will not typecheck because f and g are free. Here is a definition for *oper* that will type check and function appropriately:

$$\mathit{oper}\ f\ g\ x = \mathit{Node}\ (f\ x)\ [\mathit{oper}\ f\ g\ x' \mid x' \leftarrow g\ x]$$

If we replace *oper* with *unfold*, we get a general unfold operation for trees of any type:

$$\mathit{unfold}\ f\ g\ x = \mathit{Node}\ (f\ x)\ [\mathit{unfold}\ f\ g\ x' \mid x' \leftarrow g\ x]$$

and we an redefine *oper* as *oper1*:

$$\mathit{oper1} = \mathit{unfold}\ \mathit{id}\ \mathit{trans}$$

Following are some test cases for *trans*, *eval*, *oper* and *oper1*

$$\begin{array}{l}
\mathit{test}_0 :: \mathit{Expr}\ \mathit{Int} \\
\mathit{test}_0 = (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2)) \\
\mathit{test}_1 :: \mathit{Expr}\ \mathit{Int} \\
\mathit{test}_1 = (\mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))\ (\mathit{Val}\ 3)) \\
\mathit{test}_2 :: \mathit{Expr}\ \mathit{Int} \\
\mathit{test}_2 = (\mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))\ (\mathit{Add}\ (\mathit{Val}\ 3)\ (\mathit{Val}\ 4)))
\end{array}$$

4 Functors, Algebras and Catamorphsisms

$$\mathbf{module}\ \mathit{Cata}\ \mathbf{where}$$

The catamorphism and fold depend on the definition of the standard datatype least fixpoint. *Fix* type defines a template for fixed point data tyeps. Note that the *In* constructor is required by `Haskell` to create instances of *Fix*. The *out* function is effectively the opposite of the *In* constructor and pulls the encapsulated data structure out of the constructor.

$$\mathbf{newtype}\ \mathit{Fix}\ f = \mathit{In}\ (f\ (\mathit{Fix}\ f))$$

$$\begin{array}{l}
\mathit{out} :: \mathit{Fix}\ f \rightarrow f\ (\mathit{Fix}\ f) \\
\mathit{out}\ (\mathit{In}\ x) = x
\end{array}$$

Now we add the Algebra and Co-Algebraic notation. The original type for fold:

$$\mathit{fold} :: (\mathit{Functor}\ f) \Rightarrow (f\ a \rightarrow a) \rightarrow \mathit{Fix}\ f \rightarrow a$$

becomes:

$$\mathit{fold} :: (\mathit{Functor}\ f) \Rightarrow (\mathit{Algebra}\ f\ a) \rightarrow \mathit{Fix}\ f \rightarrow a$$

using the type classes *Functor* and *Algebra*. Even simpler, we can use the (Co)AlgebraConstructor class:

```
pCata :: (AlgebraConstructor f a) ⇒ Fix f → a
pCata = φ ∘ fmap pCata ∘ out
```

Informally, we can define *pCata* as:

```
pCata ≡ polytypic Catamorphism
```

Note the strong similarity between the polymorphic Catamorphism and the polymorphic fold, just above. *fold asks for* function ϕ ($F\ a \to a$) as an argument while *pCata knows* ϕ.

We now take these concepts and encode them as `Haskell` type classes and types. First, define *Algebra*, *CoAlgebra*, *AlgebraConstructor* and *CoAlgebraConstructor*:

```
type Algebra f a = f a → a
type CoAlgebra f a = a → f a

class Functor f ⇒ AlgebraConstructor f a
    where φ :: Algebra f a

class Functor f ⇒ CoAlgebraConstructor f a
    where psi :: CoAlgebra f a
```

We now redefine *fold* using *Functor* and *Algebra*. This is very similar to Duponcheel.

```
    -- fold :: (Functor f) =¿ (f a -¿ a) -¿ Fix f -¿ a
fold :: (Functor f) ⇒ (Algebra f a) → Fix f → a
fold g = g ∘ fmap (fold g) ∘ out
```

We can similarly define *pCata* using the *AlgebraConstructor* type class:

```
pCata :: (AlgebraConstructor f a) ⇒ Fix f → a
pCata = φ ∘ fmap pCata ∘ out
```

Finally, we re-define the denotational semantics for the simple expression language that we began with in `FoldUnfold.lhs` using, *Algebra*, datatype fixpoint and polytypic fold. Note that it is a little more complex because we carefully maintained generality over the specific numeric type used for constant values. The type, E, requires two type arguments, and everywhere we have to add the constraint (*Num a*).

```
data (Num numType) ⇒ E numType e = Val numType
                                 | Add e e

type Expr numType = Fix (E numType)

instance (Num a) ⇒ Functor (E a) where
    fmap f (Val n) = Val n
    fmap f (Add e1 e2) = Add (f e1) (f e2)

    -- combine :: (Num a) =¿ (E a) a -¿ a
  combine :: (Num a) ⇒ Algebra (E a) a
  combine (Val n) = n
```

$combine\ (Add\ n1\ n2) = (n1 + n2)$

instance $(Num\ a) \Rightarrow (AlgebraConstructor\ (E\ a)\ a)$ **where**
    $\phi = combine$

$evalExpr :: (Num\ a) \Rightarrow (Expr\ a) \rightarrow a$
$evalExpr = fold\ combine$

We can use *evalExprCata* to get the same result with much less work:

$evalExprCata :: (Num\ a) \Rightarrow (Expr\ a) \rightarrow a$
$evalExprCata = pCata$

$test_1$ is a sample expression for evaluation: $(3 + ((1 + 10) + 4))$

$test_1 = (In\ (Add\ (In\ (Val\ 3))$
    $(In\ (Add\ (In\ (Add\ (In\ (Val\ 1))$
        $(In\ (Val\ 10))))$
    $(In\ (Val\ 4))))))$

To combine with Duponcheel, we would like to compose our semantic evaluation functions into a single function. For this, we will use a *Sum* type defined exactly as in Duponcheel:

newtype $Sum\ f\ g\ x = S\ (Either\ (f\ x)\ (g\ x))$

$unS\ (S\ x) = x$

and show that *Sum* types are members of the *Functor* and *AlgebraConstructor* classes so we can use *pCata* as our evaluation function.

instance $(Functor\ f, Functor\ g) \Rightarrow Functor\ (Sum\ f\ g)$
    where $fmap\ h\ (S\ (Left\ x)) = S\ (Left\ \$\ fmap\ h\ x)$
        $fmap\ h\ (S\ (Right\ x)) = S\ (Right\ \$\ fmap\ h\ x)$

Here, using catamorphism instead of just fold, is a big win. the definition:

$\phi = either\ \phi\ \phi \circ unS$

uses ϕ to refer to 3 different implementations ϕ. We define ϕ for the *Sum* type, using the ϕ of each of the types we are *Sum*'ing. This is the result of using the *Algebra* type class synonym rather than defining a separately named ϕ for each term.

instance $(AlgebraConstructor\ f\ a, AlgebraConstructor\ g\ a) \Rightarrow$
    $AlgebraConstructor\ (Sum\ f\ g)\ a$
  -- phi :: Algebra fa == f a -¿ a
      where $\phi = either\ \phi\ \phi \circ unS$

As an example we extend Expressions defined above to include multiplication. I use this simple extension for brevity. For *any* extension, all we need to do is:

1. Define a new type for the extension
2. Show the extension is belongs to Functor and AlgebraConstructor
3. Define a new type, using Sum, to combine the original type with the extension type.

And, we have an evaluator for the more complex type, without modifying the base type and without defining any interaction between the two types.

$\mathbf{data}\ (\mathit{Num}\ \mathit{numType}) \Rightarrow M\ \mathit{numType}\ e = \mathit{Times}\ e\ e$

$\mathbf{instance}\ (\mathit{Num}\ a) \Rightarrow \mathit{Functor}\ (M\ a)\ \mathbf{where}$
$\quad \mathit{fmap}\ f\ (\mathit{Times}\ e1\ e2) = \mathit{Times}\ (f\ e1)\ (f\ e2)$

$\mathbf{instance}\ (\mathit{Num}\ a) \Rightarrow (\mathit{AlgebraConstructor}\ (M\ a)\ a)\ \mathbf{where}$
$\quad \phi\ (\mathit{Times}\ x\ y) = x * y$

$\mathbf{data}\ \mathit{BigE}\ n\ a = \mathit{BE}\ (\mathit{Sum}\ (E\ n)\ (M\ n)\ a)$

$\mathbf{type}\ \mathit{BigExpr}\ \mathit{numType} = \mathit{Fix}\ (\mathit{BigE}\ \mathit{numType})$

Just a bit of boiler plate stuff. Since `Haskell` requires that the new type, *BigE*, use a data constructor (named, *BE*, here); we have to show that this type is a member of *Functor* and *AlgebraConstructor* by simply ripping off this data constructor.

$\mathbf{instance}\ (\mathit{Num}\ a) \Rightarrow \mathit{Functor}\ (\mathit{BigE}\ a)\ \mathbf{where}$
$\quad \mathit{fmap}\ f\ (\mathit{BE}\ x) = \mathit{BE}\ (\mathit{fmap}\ f\ x)$

$\mathbf{instance}\ (\mathit{Num}\ a) \Rightarrow (\mathit{AlgebraConstructor}\ (\mathit{BigE}\ a)\ a)\ \mathbf{where}$
$\quad \phi\ (\mathit{BE}\ x) = \phi\ x$

$\mathit{evalBigExpr} :: (\mathit{Num}\ a) \Rightarrow (\mathit{BigExpr}\ a) \rightarrow a$
$\mathit{evalBigExpr} = \mathit{pCata}$

We can do the same thing with fold, but is a bit more clumsy. Now we must explicitly define the function to fold'ed over the sum.

The class *AlgebraConstructor* (and then catamorphism) are able to eliminate this step, since this function is already known, with the name ϕ.

$\mathit{unBE}\ (\mathit{BE}\ x) = x$

$\mathit{combineM} :: (\mathit{Num}\ a) \Rightarrow (M\ a)\ a \rightarrow a$
$\mathit{combineM}\ (\mathit{Times}\ x\ y) = x * y$

$\mathit{summedCombine} :: (\mathit{Num}\ a) \Rightarrow (\mathit{BigE}\ a)\ a \rightarrow a$
$\mathit{summedCombine} = \mathit{either}\ \mathit{combine}\ \mathit{combineM} \circ \mathit{unS} \circ \mathit{unBE}$

$\mathit{evalBigExprFold} :: (\mathit{Num}\ a) \Rightarrow (\mathit{BigExpr}\ a) \rightarrow a$
$\mathit{evalBigExprFold} = \mathit{fold}\ \mathit{summedCombine}$

5 Usage

This file is a template for transforming literate script into LaTeXand is not actually a `Haskell` interpreter implementation. Each section in this file is a separate module that can be loaded individually for experimentation.

Note that the interpreters have been developed under GHC and some require turning on the Glasgow Extensions. Your mileage may vary if you're using HUGS.

To build a LaTeXdocument from the interpreter files, use:

```
lhs2TeX --math Hutton.lhs > Hutton.tex
```

and run LaTeXon the result. The individual interpreters cannot be transformed to LaTeXdirectly.