Deriving Combinator Implementations

Lecture 4,
Designing and Using Combinators
John Hughes

Can We Derive Combinators from Specifications?

What sort of specifications, what sort of derivations?

- *Equational reasoning* is convenient with functional programs.
- *Equational specifications* (algebraic specifications) are directly useful for equational reasoning.

We work from equational specifications.

Example: Specifying Lists

We specify an abstract *sequence* type (lists in disguise), with operations

```
nil :: Seq a
unit :: a -> Seq a
cat :: Seq a -> Seq a -> Seq a
list :: Seq a -> [a]
```

List is an *observer*; any abstract data type must have observations (otherwise we can represent it by ()).

Axiomatising Lists

We take the following equational axioms:

```
nil `cat` xs = xs = xs `cat` nil
(xs`cat`ys)`cat`zs = xs`cat`(ys`cat`zs)
list nil = []
list (unit x`cat`xs) = x : list xs
```

These axioms are *complete*, in the sense that they define the value of every ground observation.

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq `cat` Seq
data Seq a = Nil | Unit a | Seq a `Cat` Seq a
```

Use the laws to derive an interpreter:

list (Unit x)

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq `cat` Seq
data Seq a = Nil | Unit a | Seq a `Cat` Seq a
```

Use the laws to derive an interpreter:

list (Unit x) = list (unit x)

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)      = list (unit x)
                   = list (unit x ‘cat‘ empty)
```

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)      = list (unit x)
                   = list (unit x ‘cat‘ empty)
                   = x : list empty
```

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)      = list (unit x)
                   = list (unit x ‘cat‘ empty)
                   = x : list empty
                   = x : []
```

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)      = x : []
list (xs ‘Cat‘ ys)
```

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)      = x : []
list (xs ‘Cat‘ ys) = list (xs ‘cat‘ ys)
```

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)      = x : []
list ((xs ‘Cat‘ ys) ‘Cat‘ zs) = list ((xs ‘cat‘ ys) ‘cat‘ zs)
```

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)          = x : []
list ((xs ‘Cat‘ ys) ‘Cat‘ zs) = list ((xs ‘cat‘ ys) ‘cat‘ zs)
                              = list (xs ‘cat‘ (ys ‘cat‘ zs))
```

Strategy 1: A Term Implementation

Represent sequences by the *syntax* of Seq terms:

```
Seq := nil | unit E | Seq ‘cat‘ Seq
data Seq a = Nil | Unit a | Seq a ‘Cat‘ Seq a
```

Use the laws to derive an interpreter:

```
list (Unit x)          = x : []
list ((xs ‘Cat‘ ys) ‘Cat‘ zs) = list ((xs ‘cat‘ ys) ‘cat‘ zs)
                              = list (xs ‘cat‘ (ys ‘cat‘ zs))
                              = list (xs ‘Cat‘ (ys ‘Cat‘ zs))
```

Strategy 1: A Term Implementation

The complete interpreter is:

```
list Nil = []
list (Unit x) = [x]
list (Nil ‘Cat‘ xs) = list xs
list (Unit x ‘Cat‘ xs) = x : list xs
list ((xs ‘Cat‘ ys) ‘Cat‘ zs) = list (xs ‘Cat‘ (ys ‘Cat‘ zs))
```

But we can do better…

Strategy 2: A Simplified Term Implementation

The laws can be used to *simplify* terms.

Claim: Every Seq term can be simplified to the form:

```
Sseq ::= nil | unit E `cat` Sseq
data Seq a = Nil | a `UnitCat` (Seq a)
```

Strategy 2: A Simplified Term Implementation

```
Sseq ::= nil | unit E `cat` Sseq
data Seq a = Nil | a `UnitCat` (Seq a)
```

Operations must convert simplified arguments to simplified results.

```
(x `UnitCat` xs) `cat` ys
```

Strategy 2: A Simplified Term Implementation

```
Sseq ::= nil | unit E `cat` Sseq
data Seq a = Nil | a `UnitCat` (Seq a)
```

Operations must convert simplified arguments to simplified results.

```
(x `UnitCat` xs) `cat` ys        = (unit x `cat` xs) `cat` ys
```

Strategy 2: A Simplified Term Implementation

Sseq ::= nil | unit E ‘cat‘ Sseq
data Seq a = Nil | a ‘UnitCat‘ (Seq a)

Operations must convert simplified arguments to simplified results.

(x ‘UnitCat‘ xs) ‘cat‘ ys = (unit x ‘cat‘ xs) ‘cat‘ ys
= unit x ‘cat‘ (xs ‘cat‘ ys)

Strategy 2: A Simplified Term Implementation

Sseq ::= nil | unit E ‘cat‘ Sseq
data Seq a = Nil | a ‘UnitCat‘ (Seq a)

Operations must convert simplified arguments to simplified results.

(x ‘UnitCat‘ xs) ‘cat‘ ys = (unit x ‘cat‘ xs) ‘cat‘ ys
= unit x ‘cat‘ (xs ‘cat‘ ys)
= x ‘UnitCat‘ (xs ‘cat‘ ys)

Strategy 2: A Simplified Term Implementation

The complete derived definitions are:

```
nil = Nil
unit x = x ‘UnitCat‘ Nil
Nil ‘cat‘ xs = xs
(x ‘UnitCat xs) ‘cat‘ ys = x ‘UnitCat‘ (xs ‘cat‘ ys)
list nil = []
list (x ‘UnitCat‘ xs) = x : list xs
```

Strategy 2: A Simplified Term Implementation

The complete derived definitions are:

These are *true equations*, we must prove separately they are a terminating *definition*.

```
nil = Nil
unit x = x ‘UnitCat‘ Nil
Nil ‘cat‘ xs = xs
(x ‘UnitCat xs) ‘cat‘ ys = x ‘UnitCat‘ (xs ‘cat‘ ys)
list nil = []
list (x ‘UnitCat‘ xs) = x : list xs
```

Strategy 2: A Simplified Term Implementation

The complete derived definitions are:

These are *true equations*, we must prove separately they are a terminating *definition*.

```
nil = Nil
unit x = x ‘UnitCat‘ Nil
Nil ‘cat‘ xs = xs
(x ‘UnitCat xs) ‘cat‘ ys = x ‘UnitCat‘ (xs ‘cat‘ ys)
list nil = []
list (x ‘UnitCat‘ xs) = x : list xs
```

A constructive proof that all terms *can* be expressed in the simplified form!

Strategy 2: A Simplified Term Implementation

The complete derived definitions are:

These are *true equations*, we must prove separately they are a terminating *definition*.

```
nil = Nil
unit x = x ‘UnitCat‘ Nil
Nil ‘cat‘ xs = xs
(x ‘UnitCat xs) ‘cat‘ ys = x ‘UnitCat‘ (xs ‘cat‘ ys)
list nil = []
list (x ‘UnitCat‘ xs) = x : list xs
```

Just lists in disguise:
UnitCat = (:), Nil = []

A constructive proof that all terms *can* be expressed in the simplified form!

A Problem

• Evaluating

((unit x1 `cat` unit x2) `cat` unit x3) … `cat` unit xn

is *quadratic*!

• Associativity converts this to

unit x1 `cat` (unit x2 `cat` (unit x3 … `cat` unit xn))

whose evaluation is *linear*.

• But `cat` cannot apply associativity… (it doesn't "see" the inner call of cat).

Strategy 3: Context Passing Implementation

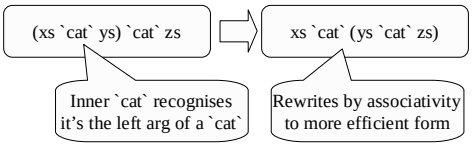
Idea: Pass `cat` a representation of its context.

(xs `cat` ys) `cat` zs

Inner `cat` recognises it's the left arg of a `cat`

Strategy 3: Context Passing Implementation

Idea: Pass `cat` a representation of its context.



Interlude: What is a Context?

A *context* C[@] is an expression "with a hole" @.

E.g. C[@] = list (@ `cat` zs)

We can "fill the hole" with an expression.

C[xs `cat` ys] = list ((xs `cat` ys) `cat` zs)

Strategy 3: Context Passing Implementation

Claim: We only need to evaluate sequences in contexts of the form

list (@ `cat` zs)

Idea: represent *contexts* as values

newtype Cxt a = ListCat (Seq a)

Represent sequences as *functions*

type Seq a = Cxt a -> [a]

Type of the observation

Strategy 3: Context Passing Implementation

ListCat zs = list (@ `cat` zs)

Let's derive some cases:

nil (ListCat zs)

Strategy 3: Context Passing Implementation

ListCat zs = list (@ ‘cat‘ zs)

Let’s derive some cases:

nil (ListCat zs) = list (nil @ `cat` zs)

Strategy 3: Context Passing Implementation

ListCat zs = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = list (nil `cat` zs)

= list zs

Strategy 3: Context Passing Implementation

ListCat zs = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = list zs

unit x (ListCat zs)

Strategy 3: Context Passing Implementation

ListCat zs = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = list zs

unit x (ListCat zs) = list (unit x `cat` zs)

Strategy 3: Context Passing Implementation

ListCat zs = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = list zs

unit x (ListCat zs) = list (unit x `cat` zs)

= x : list zs

Strategy 3: Context Passing Implementation

ListCat zs = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = list zs

unit x (ListCat zs) = x : list zs

We always seem to need list zs

Why not store list zs rather than zs?

Change variables!

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ ‘cat‘ zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs)

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = list ((xs `cat` ys) `cat` zs)

This is the *original* zs -- we must eliminate it!

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = list ((xs `cat` ys) `cat` zs)

= list (xs `cat` (ys `cat` zs))

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = list ((xs `cat` ys) `cat` zs)

= list (xs `cat` (ys `cat` zs))

= xs (ListCat (list (ys `cat` zs)))

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = list ((xs `cat` ys) `cat` zs)

= list (xs `cat` (ys `cat` zs))

= xs (ListCat (list (ys `cat` zs)))

= xs (ListCat (ys (ListCat (list zs))))

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ ‘cat‘ zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = list ((xs `cat` ys) `cat` zs)
= list (xs `cat` (ys `cat` zs))
= xs (ListCat (list (ys `cat` zs)))
= xs (ListCat (ys (ListCat zs)))

zs reintroduced

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = xs (ListCat (ys (ListCat zs)))

list xs

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = xs (ListCat (ys (ListCat zs)))

list xs = list (xs `cat` nil)

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = xs (ListCat (ys (ListCat zs)))

list xs = list (xs `cat` nil)
= xs (ListCat (list nil))

Strategy 3: Context Passing Implementation

ListCat (list zs) = list (@ `cat` zs)

Let’s derive some cases:

nil (ListCat zs) = zs

unit x (ListCat zs) = x : zs

Replace list zs by zs

(xs `cat` ys) (ListCat zs) = xs (ListCat (ys (ListCat zs)))

list xs = list (xs `cat` nil)
= xs (ListCat (list nil))
= xs (ListCat [])

Strategy 3: Context Passing Implementation

Recall definition of Cxt:

newtype Cxt a = ListCat (Seq a)

A **newtype** is unnecessary here: we can just drop the constructor ListCat.

Strategy 3: Context Passing Implementation

Collected definitions:

```
type Seq a = [a] -> [a]
nil zs = zs
unit x zs = x : zs
(xs `cat` ys) zs = xs (ys zs)
list xs = xs []
```

A sequence is a function that *prepends* its elements to a list.

Evaluation of expressions is *linear* time.

Summary of Strategies

Strategy 1: represent terms by syntax, use laws to derive an interpreter.

Strategy 2: represent terms by syntax of *simplified forms*, use laws to derive definitions which provide a *constructive proof* that every expression can be simplified.

Strategy 3: represent terms by *functions from context to observations*, use laws to derive definitions which make context-sensitive optimisations.

So?

Can we apply this to implement DSELs?

YES!

We show how to derive a *monad transformer*.

Specifying Monads

All monads should satisfy the *monad laws*:

```
return x >>= f = f x
m >>= return = m
(m >>= f) >>= g = m >>= (\x-> f x>>=g)
```

Monad transformers should satisfy:

```
lift (return x) = return x
lift (m >>= f) = lift m >>= (\x->lift (f x))
```

Specifying Run

run performs actions, as they are produced.

```
run (lift m >>= f) = m >>= \x->run (f x)
run (return x) = return x
```

Specifying Failure

failure and handle form a monoid:

```
failure `handle` m = m
m `handle` failure = m
(x `handle` y) `handle` z = x `handle` (y `handle` z)
```

Failure/Monad Interaction

These are the key properties that determine how failures behave:

```
failure >>= f = failure
return x ‘handle‘ h = return x
(lift a >>= f) ‘handle‘ h = lift a >>= \x->f x ‘handle‘ h
```

Handler discarded on success

Commit to actions once they cannot fail

Failure/Monad Interaction

These are the key properties that determine how failures behave:

```
failure >>= f = failure
return x ‘handle‘ h = return x
(lift a >>= f) ‘handle‘ h = lift a >>= \x->f x ‘handle‘ h
```

Compare *backtracking*:

```
(a ‘handle‘ b) >>= f = (a>>=f) ‘handle‘ (b>>=f)
```

B can be chosen even if f fails later

Missing Laws

Note two laws we *don’t* have:

(a ‘handle‘ b) >>= f = ???

(a >>= f) ‘handle‘ b = ???

We cannot move handlers in or out through bind: has implications for the implementation.

How Do We Choose Simplified Forms?

- Guess a suitable set (small)
- Apply all operators, try to simplify back to the chosen forms
- If you fail, add new forms to the set and repeat!

First stab

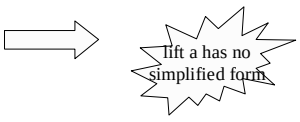
return x
failure

How Do We Choose Simplified Forms?

- Guess a suitable set (small)
- Apply all operators, try to simplify back to the chosen forms
- If you fail, add new forms to the set and repeat!

First stab

return x
failure

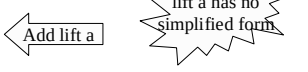


How Do We Choose Simplified Forms?

- Guess a suitable set (small)
- Apply all operators, try to simplify back to the chosen forms
- If you fail, add new forms to the set and repeat!

First stab

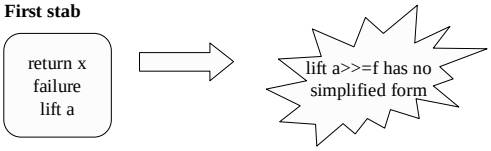
return x
failure
lift a



How Do We Choose Simplified Forms?

- Guess a suitable set (small)
- Apply all operators, try to simplify back to the chosen forms
- If you fail, add new forms to the set and repeat!

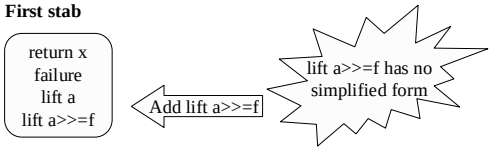
First stab



How Do We Choose Simplified Forms?

- Guess a suitable set (small)
- Apply all operators, try to simplify back to the chosen forms
- If you fail, add new forms to the set and repeat!

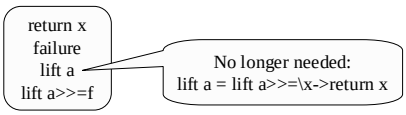
First stab



How Do We Choose Simplified Forms?

- Guess a suitable set (small)
- Apply all operators, try to simplify back to the chosen forms
- If you fail, add new forms to the set and repeat!

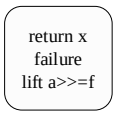
First stab



How Do We Choose Simplified Forms?

- Guess a suitable set (small)
- Apply all operators, try to simplify back to the chosen forms
- If you fail, add new forms to the set and repeat!

First stab



A complete set!

All expressions can be simplified to one of these forms.

Simplified Term Implementation

```
data Failure m a =
        Return a | Failure |
        LiftBind (m a) (a -> Failure m a)
```

return x
failure
lift a>>=f

```
return x = Return x
failure = Failure
lift a = LiftBind a Return
Return x>>=f = f x
Failure>>=f = Failure
LiftBind a f>>=g = LiftBind a (\x->f x>>=g)
Return x`handle`h = Return x
Failure`handle`h = h
LiftBind a f`handle`h = LiftBind a (\x->f x`handle`h)
```

What About Context Passing?

The following contexts suffice:

```
C[@] ::= run @
       | C[@>>=f]
       | C[@`handle`h]
```

We choose a representation

Stack of continuations and exception handlers

```
data Cxt ans a = Run (a->ans)
               | forall b. Bind (Cxt ans b) (a -> ans)
               | Handle (Cxt ans a) ans
```

Summary So Far…

- We've seen strategies for deriving *term-based* and *context-passing* implementations from an algebraic specification.
- We've seen them applied to a simple ADT and a monad transformer.
- Now for a real application…

Prettyprinting Library

Reminder: A *pretty-printer* for binary trees:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

```
prettyTree Leaf = text "Leaf"
prettyTree (Node a left right) =
   text ("(Node "++show a++" ")
     <> sep [prettyTree left, prettyTree right]
     <> text ")"
```

Example: outputs

```
(Node 2 (Node 1 Leaf Leaf)
        (Node 1 Leaf Leaf))
```

Prettyprinting Operations

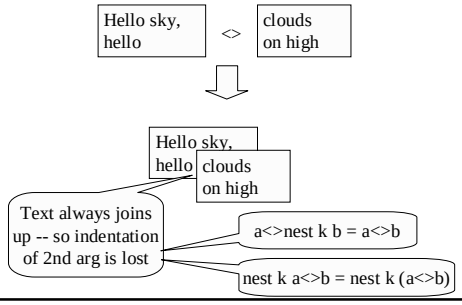
pretty :: Doc -> String

text :: String -> Doc	*literal string*
(<>), ($$) :: Doc -> Doc -> Doc	*horizontal and vertical composition*
nest :: Integer -> Doc -> Doc	*indentation*
sep :: [Doc] -> Doc	*alternative layouts*

sep [d1…dn] = d1<+>…<+>dn *or* d1 $$ … $$ dn

d1<+>d2 = d1<>text " "<>d2

Horizontal Composition



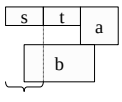
Interesting Laws

(a<>b)<>c = a<>(b<>c)
(a$$b)$$c = a$$(b$$c)
(a$$b)<>c = a$$(b<>c)
(a<>b)$$c ≠ a<>(b$$c)

This is what makes prettyprinting difficult!

Specialised Laws

(text (s++t)<>a)$$b
=
text s<>((text t<>a)$$nest(-length s)b



length s

sep [text (s++t)<>d1,…,dn]
=
text s<>sep[text t<>d1,nest(-k)d2…nest(-k)dn]
where k = length s

Reduces search: lets us generate output before exploring alternatives

Results

- The current implementation uses a simplified-term based representation, with many optimisations thanks to the algebra.
- The datatype and code is complex and impenetrable: could not be invented by informal methods.
- First (informal) implementation was buggy and slow: both performance and behaviour much improved by a formal approach.
- Extended by Simon Peyton-Jones, using same formal methods.
- Now used for all pretty-printers in GHC, hbc, …

Summary

- Algebraic specifications and equational reasoning are a good match.
- We have three standard strategies (term based, simplified term based, context passing) for deriving implementations from algebraic specifications.
- Methods are good enough to develop real libraries in widespread use.