Lecture 1: Folds and Unfolds

We'll use Haskell-like syntax, but interpret in SET rather than CPOL: all functions are total (except perhaps where stated), distinguish data from codata.

Start with (finite) lists of integers

<u>data</u> IntList
  = NilI
  | ConsI Int IntList

sum :: IntList → Int
sum NilI = 0
sum ConsI a x = ...

Many functions share sum's pattern of computation. So, abstract it – ho fns.

$fold_{IntList} :: \beta \to (Int \to \beta \to \beta) \to IntList \to \beta$
$fold_{IntList}\ e\ f\ NilI = e$
$fold_{IntList}\ e\ f\ (ConsI\ a\ x) = f\ a\ (fold\ e\ f\ x)$
$sum = fold_{IntList}\ 0\ (+)$

Many datatypes share same form of definition, so abstract it – generics.

<u>data</u> $Mu_1\ f = In\ (f\ (Mu_1\ f))$
<u>data</u> IntListF α = NilI | ConsI Int α
<u>type</u> IntList = Mu_1 IntListF

Not all f's work as parameters to Mu_1. Characterise them, as functors.

class Functor f where
$\quad$ fmap :: $(\alpha \to \beta) \to f\,\alpha \to f\,\beta$
with laws
$\quad$ fmap id $=$ id
$\quad$ fmap $(f \circ g)$ $=$ fmap f $\circ$ fmap g
Fold follows the shape of the data — "program structure follows data structure". The shape parameter f determines the datatype, but also (via fmap) the fold

$\quad$ $fold_1$ :: Functor f $\Rightarrow$ $(f\,\beta \to \beta) \to Mu_1\,f \to \beta$
$\quad$ $fold_1\,\varphi\,(In\,x) = \varphi\,(fmap\,(fold_1\,\varphi)\,x)$
Hence $fold_{IntList}$ as instance of $fold_1$.
Another example: naturals.
$\quad$ data Maybe α = Just α | Nothing
$\quad$ instance Functor Maybe where...
$\quad$ type Nat = Mu_1 Maybe
$\quad$ $fold_1$ :: $(Maybe\,\beta \to \beta) \to Nat \to \beta$
(What does this do? Nicer form?)
All very well, but this doesn't really capture the essence of "container datatypes" — Ints are hardwired in IntList, and can't define map. List α

instance Functor IntListF where...

So abstract also over element type.

class Bifunctor f **where**

$\quad$ bimap :: $(\alpha \to \gamma) \to (\beta \to \delta) \to f\,\alpha\,\beta \to f\,\gamma\,\delta$

with laws

$\quad$ bimap id id = id

$\quad$ bimap f g ∘ bimap h k = bimap (f∘h) (g∘k)

eg

$\quad$ **data** ListF $\alpha\,\beta$ = NilF | ConsF $\alpha\,\beta$

$\quad$ **instance** Bifunctor ListF **where** ...

then

$\quad$ **data** Mu f α = In (f α (Mu f α)) {in ::}

$\quad$ **type** List α = Mu (ListF α)

As before

$\quad$ fold :: Bifunctor f ⇒ (f $\alpha\,\beta \to \beta$) → Mu f $\alpha \to \beta$

$\quad$ fold φ (In x) = φ (bimap id (fold φ) x)

but now we can also define

$\quad$ map :: Bifunctor f ⇒ $(\alpha \to \beta)$ → Mu f α → ...

$\quad$ map h (In x) = In (bimap h (map h) x)

In fact, map h is an instance of fold. Also, it makes Mu f an instance of Functor!

The definition of fold is an equation about its behaviour ("evaluation"):

$$\text{fold } \varphi \cdot \text{In} = \varphi \cdot \text{bimap id } (\text{fold } \varphi)$$

In fact fold φ is the <u>unique</u> solution to this equation:

$$h = \text{fold } \varphi \Leftrightarrow h \cdot \text{In} = \varphi \cdot \text{bimap id } h$$

This is the <u>universal property</u> of fold. Evaluation is a special case, obtained by letting $h = \text{fold } \varphi$. Another special case arises by letting $h = \text{id}$ (and $\varphi = \text{In}$ — calculate "reflection".

Third, we get the <u>fusion law</u>

$$h \cdot \text{fold } \varphi = \text{fold } \psi \Leftarrow \ldots$$

— in fact, there is an exact version too. And as a special case of this, we get <u>fold-map fusion</u>:

$$\text{fold } \varphi \cdot \text{map } h = \text{fold } (\varphi \cdot \text{bimap } h \text{ id})$$

Of course, it all dualizes nicely.

```
codata Nu f α = Out° {out :: f α (Nu f α)}
```

Operationally, this datatype contains both finite and infinite datastructures. Structural recursion may not be well-founded; but corecursion works totally.

```
unfold :: Bifunctor f ⇒ (β → f α β) → β → Nu f α
unfold φ = Out° ∘ bimap id (unfold φ) ∘ φ
```

Eg map is ~~also~~ an unfold

```
map f = unfold (bimap f id ∘ out)
```

for exaple, lists as codata:

```
type Colist α = Nu ListF α
```

So range generates finite or infinite lists:

```
range :: (Int, Int) → Colist Int
range = unfold next where
    next (m,n) | m==n = NilF
               | otherwise = ConsF m (m+1, n)
```

Universal property

$$h = \mathit{unfold}\ \varphi \Leftrightarrow \mathit{out} \circ h = \mathit{bimap}\ \mathit{id}\ h \circ \varphi$$

reflection, unfold fusion, map unfold fusion.

Maybe present binary trees too.