Lecture 11: Primitive (Co-)Recursion

One of the first examples of recursion any programmer learns is the factorial function:

$$\text{fact zero} = 1$$
$$\text{fact}\,(\text{succ}\,n) = \text{succ}\,n \times \text{fact}\,n$$

This is not in the form of a fold on naturals, because the recursive case depends on n as well as fact n.
In fact, this is an instance of primitive recursion, whereas folds are iterations.

We can capture this pattern of recursion in the same manner, though; it was dubbed paramorphism by Meertens. The key fact is that (when $\tau = \text{fix}\, F$)
$\text{para}_\tau\, f :: \tau \to \alpha$ uses a body $f :: F(\alpha \times \tau) \to \alpha$, with every recursive result of type α tupled with the substructure ($:: \tau$) whence it came. We define

$$\text{para}_\tau f = \text{fst} \cdot \text{fold}_\tau (f \vartriangle (\text{in}_\tau \cdot F\,\text{snd}))$$

It is easy to show, using the universal property, that

$$\text{snd} \cdot \text{fold}_\tau (f \vartriangle (\text{in}_\tau \cdot F\,\text{snd})) = \text{fold}_\tau\, \text{in}_\tau = id$$

and so

$$\text{fold}_\tau (f \vartriangle (\text{in}_\tau \cdot F\,\text{snd})) = \text{para}_\tau\, f \vartriangle id$$

Paramorphisms generalize folds, since

$fold_\tau\, f = para_\tau\,(f \cdot F\, fst)$

simply by ignoring the extra information.

and in particular, $id = para_\tau\,(in_\tau \cdot F\, fst)$.

Paramorphisms enjoy a universal property

$h = para_\tau\, f$

$\equiv\ h = fst \cdot fold_\tau\,(f \vartriangle (in_\tau \cdot F snd))$

$\equiv\ h \vartriangle id = (fst \cdot fold_\tau\,(f \vartriangle (in_\tau \cdot F snd))) \vartriangle id$

$\equiv\ h \vartriangle id = fold_\tau\,(f \vartriangle (in_\tau \cdot F snd))$

$\equiv\ (h \vartriangle id) \cdot in_\tau = f \vartriangle (in_\tau \cdot F snd) \cdot F\,(h \vartriangle id)$

$\equiv\ h \cdot in_\tau = f \cdot F\,(h \vartriangle id)$

from which we get a fusion law

$h \cdot para_\tau f = para_\tau\, g \Leftarrow h \cdot f = g \cdot F\,(h \times id)$

More surprisingly, any function (from τ)

is a paramorphism:

$h = para_\tau\,(h \cdot in_\tau \cdot F\, snd)$

Back to examples: $fact = para_{Nat}\, f$ where

$f = const\ 1 \triangledown (mul \cdot (id \times succ)) :: 1 + Nat \times Nat \to Nat$

The fn $del :: List\ \alpha \to 1 + \alpha \times List\ \alpha$ from lecture

is $para_{List}\,(id + f)$ where

$f(a, (inl\ (), nil)) = inr\,(a, nil)$ cons(a,y)

$f\,(a, (inr\,(b,y), x)) = $ if $a \le b$ then $inr(a,x)$ else $inr(b$

We saw at the end of lecture 6 that factorial is also a hylomorphism; this generalises to all paramorphisms.

For $\tau = \underline{\text{fix}}\, F$, let $G\beta = F(\beta \times \tau)$ and $\sigma = \underline{\text{fix}}\, G$. We define the predecessor by

$$\text{preds}_\tau = \text{unfold}_\sigma (F(\text{id} \vartriangle \text{id}) \cdot \text{out}_\tau) :: \tau \to \sigma$$

Then $\text{para}_\tau f = \text{fold}_\sigma f \cdot \text{preds}_\tau$. We can try evaluating this to dissect the σ:

$$\begin{aligned}
&\text{fold}_\sigma f \cdot \text{unfold}_\sigma (F(\text{id} \vartriangle \text{id}) \cdot \text{out}_\tau) \\
= \; & f \cdot G(\text{para}_\tau f) \cdot F(\text{id} \vartriangle \text{id}) \cdot \text{out}_\tau \\
= \; & f \cdot F(\text{para}_\tau f \times \text{id}) \cdot F(\text{id} \vartriangle \text{id}) \cdot \text{out}_\tau \\
= \; & f \cdot F(\text{para}_\tau f \vartriangle \text{id}) \cdot \text{out}_\tau
\end{aligned}$$

We can also show again using the universal property of fold, that

$$(\text{fold}_\sigma f \cdot \text{unfold}_\sigma (F(\text{id} \vartriangle \text{id}) \cdot \text{out}_\tau)) \vartriangle \text{id} = \text{fold}_\tau (f \vartriangle (\text{in}_\tau \cdot F\text{snd}))$$

In the case of naturals, $F\alpha = 1 + \alpha$, so $G\alpha = 1 + \alpha \times \text{Nat}$, and σ is (isomorphic to) List Nat. In fact, $\text{preds}_{\text{Nat}}$ as defined here gives a snoc-list of the numbers less than a given n, whereas preds from lecture 6 had less than or equal, but no succs in prod.

This can dualizes, of course. For body $f :: \alpha \to F(\alpha + \tau)$, we define the apomorphism $apo_\tau\, f :: \alpha \to \tau$ by

$apo_\tau\, f = unfold_\tau\, (f \triangledown (F\, inr \cdot out_\tau)) \cdot inl$

Informally, this is like an unfold but with the option, whenever generating a new seed, to generate a whole substructure instead. The standard reference is Vene & Uustalu, although the ideas appear in unpublished work by Vos.

The unfold-outer half is

$unfold_\tau\, (f \triangledown (F\, inr \cdot out_\tau)) \cdot inr = id$

and so

$unfold_\tau\, (f \triangledown (F\, inr \cdot out_\tau)) = apo_\tau\, f \triangledown id$

Apomorphisms generalize unfolds:

$unfold_\tau\, f = apo_\tau\, (F\, inl \cdot f)$

(never using the extra opportunity), and in particular

$apo_\tau\, (F\, inl \cdot out_\tau) = id$

The fusion law is

$apo_\tau\, f \cdot h = apo_\tau\, g \Leftarrow f \cdot h = F(h \triangledown id) \cdot g$

following from the universal property

$h = apo_\tau f \equiv out_\tau \cdot h = F(h \triangledown id) \cdot f$

Any function to a (co)datatype can be written as an apomorphism:

$h = apo_\tau (F\, inr \cdot out_\tau \cdot h)$

(Note that, as with the dual results for paramorphisms, this does not give an effective definition for h, because of the recurrence on the rhs. Thus it does not follow that paramorphisms or apomorphisms alone are computationally adequate.)

Apomorphisms are hylomorphisms too. For $\tau = fix\, F$ we let $H\beta = F(\beta + \tau)$ and $\rho = fix\, H$. Then

$apo_\tau f = fold_\rho (in_\tau \cdot F(id \triangledown id)) \cdot unfold_\rho f$

One might call the fold phase of this "collecting" or "harvesting" the result, either element by element or in chunks. The intermediate datastructure deforests:

$apo_\tau f = in_\tau \cdot F(apo_\tau f \triangledown id) \cdot f$

and we can, as before, verify that

$(fold_\rho (in_\tau \cdot F(id \triangledown id)) \cdot unfold_\rho f) \triangledown id$
$= unfold_\tau (f \triangledown (F\, inr \cdot out_\tau))$

One simple example of an apomorphism is cat – also a fold, but coiteratively it is unfold_{List} next where

$\mathrm{next}\ (\mathrm{nil}, \mathrm{nil}) = \mathrm{inl}\ ()$
$\mathrm{next}\ (\mathrm{cons}(a,x), y) = \mathrm{inr}\ (a, (x,y))$
$\mathrm{next}\ (\mathrm{nil}, \mathrm{cons}(b,y)) = \mathrm{inr}\ (b, (\mathrm{nil},y))$

This is inefficient, copying the tail y element by element; but $\mathrm{cat} = \mathrm{apo}_{List}\ \mathrm{next}'$:

$\mathrm{next}'\ (\mathrm{nil}, \mathrm{nil}) = \mathrm{inl}\ ()$
$\mathrm{next}'\ (\mathrm{cons}(a,x), y) = \mathrm{inr}\ (a, \mathrm{inl}\ (x,y))$
$\mathrm{next}'\ (\mathrm{nil}, \mathrm{cons}(b,y)) = \mathrm{inr}\ (b, \mathrm{inr}\ y)$

From lecture 8, $\mathrm{ins} :: 1 + \alpha \times \mathrm{List}\ \alpha \to \mathrm{List}\ \alpha = \mathrm{apo}_{List}\ \mathrm{comp}$:

$\mathrm{comp}\ (\mathrm{inl}\ ()) = \mathrm{inl}\ ()$
$\mathrm{comp}\ (\mathrm{inr}\ (a, \mathrm{nil})) = \mathrm{inr}\ (a, \mathrm{inr}\ \mathrm{nil})$
$\mathrm{comp}\ (\mathrm{inr}\ (a, \mathrm{cons}(b,y))) =$
  $\mathrm{if}\ a \leq b\ \mathrm{then}\ \mathrm{inr}\ (a, \mathrm{inr}\ (\mathrm{cons}(b,y)))$
  $\mathrm{else}\ \mathrm{inr}\ (b, \mathrm{inl}\ (\mathrm{inr}\ (a,y)))$

and from lectures 8 and 10, $\mathrm{lzw}\ f = \mathrm{apo}_{List}\ \mathrm{ap}$:

$\mathrm{ap}\ (\mathrm{nil}, \mathrm{nil}) = \mathrm{inl}\ ()$
$\mathrm{ap}\ (\mathrm{nil}, \mathrm{cons}(b,y)) = \mathrm{inr}\ (b, \mathrm{inr}\ y)$
$\mathrm{ap}\ (\mathrm{cons}(a,x), \mathrm{nil}) = \mathrm{inr}\ (a, \mathrm{inr}\ x)$
$\mathrm{ap}\ (\mathrm{cons}(a,x), \mathrm{cons}(b,y)) = \mathrm{inr}\ (f(a,b), \mathrm{inl}\ (x,y))$

$\mathrm{apo}_{List} :: (\beta \to 1 + \alpha \times (\beta + [\alpha])) \to \beta \to [\alpha]$