Recursion schemes

HaskHEL meetup at Gofore

Oleg Grenrus

Futurice

2017-02-28

HaskHEL

- `/join #haskhel` on freenode
- We need talks: topics, presenters...
- `https://github.com/haskhel/events`

References

- Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire (1991)
- Data types la carte (2008)
- `recursion-schemes` package

this is literal haskell file

```
stack --resolver=nightly-2017-02-01 ghci
Prelude> :l schemes.lhs
```

to play around.

Factorial

$$
\begin{aligned}
&\textbf{data } \mathit{NatF}\ a = \mathit{ZeroF} \mid \mathit{SuccF}\ a \textbf{ deriving } (\mathit{Show}, \mathit{Functor})\\
&\textbf{type instance } \mathit{Base}\ \mathit{Natural} = \mathit{NatF}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Recursive}\ \mathit{Natural} \textbf{ where}\\
&\quad \mathit{project}\ 0 = \mathit{ZeroF}\\
&\quad \mathit{project}\ n = \mathit{SuccF}\ (n - 1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{factorial} :: \mathit{Natural} \to \mathit{Natural}\\
&\mathit{factorial} = \mathit{para}\ \mathit{alg} \textbf{ where}\\
&\quad \mathit{alg}\ \mathit{ZeroF} \qquad\qquad\quad = 1\\
&\quad \mathit{alg}\ (\mathit{SuccF}\ (n, m)) = (1 + n) * m
\end{aligned}
$$

Why not explict recursion?

$$
\begin{aligned}
&\mathit{factorial\_1} :: \mathit{Natural} \to \mathit{Natural} \\
&\mathit{factorial\_1}\ 0 = 1 \\
&\mathit{factorial\_1}\ n = n * \mathit{factorial}\ (n - 1)
\end{aligned}
$$

$$\mathit{factorial\_2} :: \mathit{Natural} \rightarrow \mathit{Natural}$$
$$\mathit{factorial\_2}\ n = \mathit{product}\ [1 \mathinner{..} n]$$

For more see *The Evolution of a Haskell Programmer*

Spot a bug

$$
\begin{array}{l}
\mathbf{newtype}\ State\ s\ a = State\ \{runState :: s \to (a, s)\} \\
\mathbf{instance}\ Monad\ (State\ s)\ \mathbf{where} \\
\quad return\ x = State\ \$\ \lambda s \to (x, s) \\
\quad m \gg\!= k = State\ \$\ \lambda s_0 \to \\
\qquad \mathbf{let}\ (x_1, s_1) = runState\ m\ s_0 \\
\qquad\qquad (x_2, s_2) = runState\ (k\ x_1)\ s_0 \quad \text{-- should be } s_1 \\
\qquad \mathbf{in}\ (x_2, s_2)
\end{array}
$$

type-checker cannot help here…

Spot another bug

```haskell
type VarName = String
data Expr = Var VarName
          | Lit Int
          | Add Expr Expr
          | Mul Expr Expr
  deriving (Show)

subst :: VarName -> Int -> Expr -> Expr
subst n x (Var n')
    | n == n'          = Lit x
    | otherwise        = Var n'
subst n x (Lit l)      = Lit l
subst n x (Add a b)    = Add (subst n x a) (subst n x b)
subst n x (Mul a b)    = Mul (subst n x a) b    -- no subst
```

In our code base atm

```
% git grep 'cata' | wc -l
3
```

Why? You *had* to write boilerplate.

Outsource writing of boilerplate

With *TemplateHaskell*

import *Data.Functor.Foldable.TH*

a single magic spell

makeBaseFunctor ''*Expr*

generates

data *ExprF* $= \ldots$
type instance *Base Expr* $=$ *ExprF*
instance *Recursive Expr* **where** *project* $= \ldots$
instance *Corecursive Expr* **where** *embed* $= \ldots$

$$
\begin{aligned}
&\mathit{subst}_2 :: \mathit{VarName} \to \mathit{Int} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{subst}_2\ n\ x = \mathit{cata}\ \mathit{alg}\ \mathbf{where} \\
&\quad \mathit{alg}\ (\mathit{VarF}\ n') \mid n \equiv n' = \mathit{Lit}\ x \\
&\quad \mathit{alg}\ x \qquad\qquad\qquad\quad\;\; = \mathit{embed}\ x
\end{aligned}
$$

The following identity holds:

$$
\mathit{cata}\ \mathit{embed} = \mathit{id}
$$

Decomposing cata

$$\textbf{data}\ \mathit{ListF}\ a\ b = \mathit{NilF} \mid \mathit{ConsF}\ a\ b$$
$$\mathit{cata} :: (\mathit{ListF}\ a\ b \to b) \to [a] \to b$$

Decomposing cata: 2

$$\textbf{data}\ \mathit{ListF}\ a\ b = \mathit{NilF} \mid \mathit{ConsF}\ a\ b$$

$$\begin{aligned} \mathit{cata} &:: (\mathit{ListF}\ a\ b \to b) &&\to [a] \to b \\ \mathit{cata} &:: (\mathit{Either}\ ()\ (a, b) \to b) &&\to [a] \to b \end{aligned}$$

Decomposing cata: 3

$$\textbf{data}\ \mathit{ListF}\ a\ b = \mathit{NilF} \mid \mathit{ConsF}\ a\ b$$

$$\begin{aligned}
\mathit{cata} &:: (\mathit{ListF}\ a\ b \to b) && \to [a] \to b \\
\mathit{cata} &:: (\mathit{Either}\ ()\ (a, b) \to b) && \to [a] \to b \\
\mathit{cata} &:: (() \to b) \to ((a, b) \to b) && \to [a] \to b
\end{aligned}$$

Decomposing cata: 4

$$\textbf{data}\ \mathit{ListF}\ a\ b = \mathit{NilF} \mid \mathit{ConsF}\ a\ b$$

$$\begin{aligned}
\mathit{cata} &:: (\mathit{ListF}\ a\ b \to b) && \to [a] \to b \\
\mathit{cata} &:: (\mathit{Either}\ ()\ (a, b) \to b) && \to [a] \to b \\
\mathit{cata} &:: (() \to b) \to ((a, b) \to b) && \to [a] \to b \\
\mathit{cata} &:: b \to (a \to b \to b) && \to [a] \to b
\end{aligned}$$

Decomposing cata: 5

$$
\begin{aligned}
&\mathbf{data}\ \mathit{ListF}\ a\ b = \mathit{NilF} \mid \mathit{ConsF}\ a\ b \\
&\mathit{cata} :: (\mathit{ListF}\ a\ b \to b) && \to [a] \to b \\
&\mathit{cata} :: (\mathit{Either}\ ()\ (a, b) \to b) && \to [a] \to b \\
&\mathit{cata} :: (() \to b) \to ((a, b) \to b) && \to [a] \to b \\
&\mathit{cata} :: b \to (a \to b \to b) && \to [a] \to b \\
&\mathit{cata} :: (a \to b \to b) \to b && \to [a] \to b
\end{aligned}
$$

Decomposing cata: 6

$$\mathbf{data}\ \mathit{ListF}\ a\ b = \mathit{NilF} \mid \mathit{ConsF}\ a\ b$$

$$\begin{aligned}
\mathit{cata} &:: (\mathit{ListF}\ a\ b \to b) && \to [a] \to b \\
\mathit{cata} &:: (\mathit{Either}\ ()\ (a,b) \to b) && \to [a] \to b \\
\mathit{cata} &:: (() \to b) \to ((a,b) \to b) && \to [a] \to b \\
\mathit{cata} &:: b \to (a \to b \to b) && \to [a] \to b \\
\mathit{cata} &:: (a \to b \to b) \to b && \to [a] \to b
\end{aligned}$$

$$\mathit{cata} = \mathit{foldr}$$

Another way to look on cata

$$cata\ f = c\ \mathbf{where}\ c = f \circ fmap\ c \circ project$$

$$list :: List\ Int$$
$$list = Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))$$

So what *cata* does?

$$\begin{aligned}
&cata\ alg\ list = \\
&\quad = c\ \$\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))) \\
&\quad = alg \circ fmap\ c \circ project\ \$\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))) \\
&\quad = alg \circ fmap\ c\ \$\ ConsF\ 1\ (Cons\ 2\ (Cons\ 3\ Nil)) \\
&\quad = alg\ \$\ ConsF\ 1\ \$\ c\ (Cons\ 2\ (Cons\ 3\ Nil)) \\
&\quad = alg\ \$\ ConsF\ 1\ \$\ alg \circ fmap\ c \circ project\ \$\ (Cons\ 2\ (Cons\ 3\ Nil))
\end{aligned}$$

$$
\begin{aligned}
&\mathit{cata}\ \mathit{alg}\ \mathit{list} = \\
&\quad \ldots \\
&\quad = \mathit{alg}\ \$\ \mathit{ConsF}\ 1\ \$\ \mathit{alg} \circ \mathit{fmap}\ c \circ \mathit{project}\ \$\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil})) \\
&\quad = \mathit{alg}\ \$\ \mathit{ConsF}\ 1\ \$\ \mathit{alg}\ \$\ \mathit{ConsF}\ 2\ \$\ c\ (\mathit{Cons}\ 3\ \mathit{Nil}) \\
&\quad = \mathit{alg}\ \$\ \mathit{ConsF}\ 1\ \$\ \mathit{alg}\ \$\ \mathit{ConsF}\ 2\ \$\ \mathit{alg}\ \$\ \mathit{ConsF}\ 3\ \$\ c\ \mathit{Nil} \\
&\quad = \mathit{alg}\ \$\ \mathit{ConsF}\ 1\ \$\ \mathit{alg}\ \$\ \mathit{ConsF}\ 2\ \$\ \mathit{alg}\ \$\ \mathit{ConsF}\ 3\ \$\ \mathit{alg}\ \mathit{NilF}
\end{aligned}
$$

so similarly as *foldr* $f\ z$ replaces list constuctors with f and z, *cata* replaces them with *alg* $(\mathit{ConsF}\ \_\ \_)$ and *alg NilF*

-morphisms

cata	fold
para	also fold
ana	unfold
apo	also unfold
hylo	refold

Monadic morphisms

$$
\begin{aligned}
cata \; &:: \; Recursive\ t \\
&\Rightarrow (Base\ t\ a \to a) \quad \to t \to a
\end{aligned}
$$

Monadic variant is not yet in `recursion-schemes`:

$$
\begin{aligned}
&cataM \\
&\quad :: \; (Recursive\ t, Traversable\ (Base\ t), Monad\ m) \\
&\quad \Rightarrow (Base\ t\ a \quad \to m\ a) \to t \to m\ a \\
&cataM\ f = (\gg\!= f) \circ cata\ (traverse\ (\gg\!= f))
\end{aligned}
$$

We can print all intermediate structs:

$$
\begin{aligned}
&ex\_1 :: IO\ () \\
&ex\_1 = cataM\ print\ expr
\end{aligned}
$$

Elm can cata too

Recursion schemes is advanced technology …of ’90s.

$$\begin{array}{l}
\textbf{type}\ \mathit{ListF}\ a\ b = \mathit{NilF} \mid \mathit{ConsF}\ a\ b \\
\mathit{listCata} : (\mathit{ListF}\ a\ b \to b) \to \mathit{List}\ a \to b \\
\mathit{listCata}\ \mathit{alg}\ l = \textbf{case}\ l\ \textbf{of} \\
\quad [\,] \qquad\quad \to \mathit{alg}\ \mathit{NilF} \\
\quad (x :: \mathit{xs}) \to \mathit{alg}\ (\mathit{ConsF}\ x\ (\mathit{listCata}\ \mathit{alg}\ \mathit{xs}))
\end{array}$$

Scala can too: `matryoshka`.

aeson-extra

Uses base functor to restrict the possible choices:

$$
\begin{aligned}
&\mathit{merge} \\
&\quad :: (\forall a.(a \to a \to a) \to \mathit{ValueF}\ a \to \mathit{ValueF}\ a \to \mathit{ValueF}\ a) \\
&\quad \to \mathit{Value} \to \mathit{Value} \to \mathit{Value}
\end{aligned}
$$

from *Data.Aeson.Extra.Merge* module

unification-fd

Uses base functor to amend the recursive type with additional construct:

$$\begin{aligned}\mathbf{data}\ \mathit{UTerm}\ t\ v &= \mathit{UVar}\ v \\ &\mid\ \mathit{UTerm}\ (t\ (\mathit{UTerm}\ t\ v))\end{aligned}$$

compare to

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

see *Control.Unification* module

live long and recurse!

Extras: IxState

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{IxState}\ i\ o\ a = \mathit{IxState}\ \{\mathit{runIxState} :: i \to (a, o)\} \\
\textbf{infixl}\ 1 \ggg\!= \\
(\ggg\!=) :: \mathit{IxState}\ s_0\ s_1\ a \to (a \to \mathit{IxState}\ s_1\ s_2\ b) \to \mathit{IxState}\ s_0\ s_2\ b \\
m \ggg\!= k = \mathit{IxState}\ \$\ \lambda s_0 \to \\
\quad \textbf{let}\ (x_1, s_1) = \mathit{runIxState}\ m\ s_0 \\
\qquad\ \ (x_2, s_2) = \mathit{runIxState}\ (k\ x_1)\ s_1 \\
\quad \textbf{in}\ (x_2, s_2) \\
\mathit{bindState} :: \mathit{State}\ s\ a \to (a \to \mathit{State}\ s\ b) \to \mathit{State}\ s\ b \\
\mathit{bindState}\ m\ k = \mathit{to}\ (\mathit{from}\ m \ggg\!= \mathit{from} \circ k)\ \textbf{where} \\
\quad \mathit{to} \quad\ \ = \mathit{State} \circ \mathit{runIxState} \\
\quad \mathit{from} = \mathit{IxState} \circ \mathit{runState}
\end{array}
$$