Slide 1

Generic programming with morphisms

29/03/04

Roman Leshchinskiy

Slide 2

WHAT IS THIS ABOUT?

$\mathbf{data}\ List\ a\ =\ Nil\ |\ Cons\ a\ (List\ a)$
$\mathbf{data}\ Tree\ a\ =\ Empty\ |\ Node\ a\ (Tree\ a)\ (Tree\ a)$

$length :: List\ a \rightarrow Int$
$length\ Nil = 0$
$length\ (Cons\ x\ xs) = 1 + length\ xs$

$size :: Tree\ a \rightarrow Int$
$size\ Empty = 0$
$size\ (Node\ x\ t\ u) = 1 + size\ t + size\ u$

What do these functions have in common?

Slide 3

WARMING UP – FUNCTORS

- the well-known map
  $map :: (a \rightarrow b) \rightarrow List\ a \rightarrow List\ b$
- a no-so-well-known map
  $mapMaybe :: (a \rightarrow b) \rightarrow Maybe\ a \rightarrow Maybe\ b$
- $List$ and $Maybe$ are functors

$\mathbf{class}\ Functor\ f\ \mathbf{where}$
$\quad fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
$\mathbf{instance}\ Functor\ List\ \mathbf{where}$
$\quad fmap = map$
$\mathbf{instance}\ Functor\ Maybe\ \mathbf{where}$
$\quad fmap = mapMaybe$
$\mathbf{instance}\ Functor\ Tree\ \mathbf{where}$
$\quad fmap = mapTree$

Slide 4

DATATYPE AS FIXPOINTS

What does this declaration mean?

$\mathbf{data}\ List\ a\ =\ Nil\ |\ Cons\ a\ (List\ a)$

- $List$ is a recursive datatype
- recursion can be expressed by fixpoints

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

- $List$ is a fixpoint of the its pretype $PreList$
  $\mathbf{type}\ List\ a\ =\ Fix\ (PreList\ a)$
- how is $PreList$ defined?

$\mathbf{data}\ PreList\ a\ l\ =\ Nil\ |\ Cons\ a\ l$
$Fix\ (PreList\ a)\ =\ Nil\ |\ Cons\ a\ (Fix\ (PreList\ a))$

Slide 5

FIXPOINTS – EXAMPLES

Trees

$$\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a\ =\ \mathit{Empty} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\\
&\mathbf{type}\ \mathit{Tree}\ a\ =\ \mathit{Fix}\ (\mathit{PreTree}\ a)\\
&\mathbf{data}\ \mathit{PreTree}\ a\ t\ =\ \mathit{Empty} \mid \mathit{Node}\ a\ t\ t
\end{aligned}$$

Natural numbers

$$\begin{aligned}
&\mathbf{data}\ \mathit{Nat}\ =\ \mathit{Zero} \mid \mathit{Suc}\ \mathit{Nat}\\
&\mathbf{type}\ \mathit{Nat}\ =\ \mathit{Fix}\ \mathit{PreNat}\\
&\mathbf{data}\ \mathit{PreNat}\ n\ =\ \mathit{Zero} \mid \mathit{Suc}\ n
\end{aligned}$$

Trick: replace all recursive occurences by a type parameter

Slide 6

WORKING WITH Fix

$$\begin{aligned}
&\mathit{length} && :: \mathit{List}\ a \to \mathit{Int}\\
& && :: \mathit{Fix}\ (\mathit{PreList}\ a) \to \mathit{Int}\\
&\mathit{length}\ (\mathit{In}\ \mathit{Nil}) && = 0\\
&\mathit{length}\ (\mathit{In}\ (\mathit{Cons}\ x\ xs)) && = 1 + \mathit{length}\ xs
\end{aligned}$$

$$\begin{aligned}
&\mathit{size} && :: \mathit{Tree}\ a \to \mathit{Int}\\
& && :: \mathit{Fix}\ (\mathit{PreTree}\ a) \to \mathit{Int}\\
&\mathit{size}\ (\mathit{In}\ \mathit{Empty}) && = 0\\
&\mathit{size}\ (\mathit{In}\ (\mathit{Node}\ x\ t\ u)) && = 1 + \mathit{size}\ t + \mathit{size}\ u
\end{aligned}$$

Not really hard!

Slide 7

FIXPOINTS – REMARKS

- The constructor In is artificial in a sense, but

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

  is illegal.
- helper functions (isomorphisms)

  $$\begin{aligned}
  &\mathit{inFix} && :: f\ (\mathit{Fix}\ f) \to \mathit{Fix}\ f\\
  &\mathit{inFix} && = \mathit{In}\\
  &\mathit{outFix} && :: \mathit{Fix}\ f \to f\ (\mathit{Fix}\ f)\\
  &\mathit{outFix}\ (\mathit{In}\ x) && = x
  \end{aligned}$$

- $\mathbf{data}\ \mathit{Fix}\ f\ =\ \mathit{In}\ (f\ (\mathit{Fix}\ f))$
  is wrong for obscure reasons related to strictness

Slide 8

PRETYPES ARE FUNCTORS!

- so far, we have only used $\mathit{PreList}\ a\ (\mathit{Fix}\ (\mathit{PreList}\ a))$
- what about e.g. $\mathit{PreList}\ a\ \mathit{Int}$?

$$\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ (\mathit{PreList}\ a)\ \mathbf{where}\\
&\quad \mathit{fmap} && :: (l \to l') \to \mathit{PreList}\ a\ l \to \mathit{PreList}\ a\ l'\\
&\quad \mathit{fmap}\ f\ \mathit{Nil} && = \mathit{Nil}\\
&\quad \mathit{fmap}\ f\ (\mathit{Cons}\ x\ l) && = \mathit{Cons}\ x\ (f\ l)
\end{aligned}$$

Operates on the second (recursive) component of Cons, not on the list elements!

$$\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ (\mathit{PreTree}\ a)\ \mathbf{where}\\
&\quad \mathit{fmap} && :: (t \to u) \to \mathit{PreTree}\ a\ t \to \mathit{PreTree}\ a\ u\\
&\quad \mathit{fmap}\ f\ \mathit{Empty} && = \mathit{Empty}\\
&\quad \mathit{fmap}\ f\ (\mathit{Node}\ x\ t\ u) && = \mathit{Node}\ x\ (f\ t)\ (f\ u)
\end{aligned}$$

Dissecting $size$

Slide 9

$$size\;(In\;Empty) = 0$$
$$size\;(In\;(Node\;x\;t\;u)) = 1 + size\;t + size\;u$$

We want to apply $size$ to every recursive subterm

$$\begin{aligned} & size\;(In\;t) = \textbf{case}\; fmap\;size\;t\;\textbf{of} \\ & \quad Empty \to 0 \\ & \quad Node\;x\;m\;n \to 1 + m + n \end{aligned}$$

Simplifying a bit

$$\begin{aligned} size &= size' \circ fmap\;size \circ outFix \\ size' &:: PreTree\;a\;Int \to Int \\ size'\;Empty &= 0 \\ size'\;(Node\;x\;m\;n) &= 1 + m + n \end{aligned}$$

Dissecting $length$

Slide 10

$$length\;(In\;Nil) = 0$$
$$length\;(In\;(Cons\;x\;xs)) = 1 + length\;xs$$

We want to apply $length$ to every recursive subterm

$$\begin{aligned} & length\;(In\;l) = \textbf{case}\; fmap\;length\;l\;\textbf{of} \\ & \quad Nil \to 0 \\ & \quad Cons\;x\;n \to 1 + n \end{aligned}$$

Simplifying

$$\begin{aligned} length &= length' \circ fmap\;length \circ outFix \\ length' &:: PreList\;a\;Int \to Int \\ length'\;Nil &= 0 \\ length'\;(Cons\;x\;n) &= 1 + n \end{aligned}$$

Generalising

Slide 11

$$\begin{aligned} size &:: Fix\;(\;PreTree\;a) \to Int \\ length &:: Fix\;(\;PreList\;a) \to Int \\ size &= size' \circ fmap\;size \circ outFix \\ length &= length' \circ fmap\;length \circ outFix \\ size' &:: PreTree\;a\;Int \to Int \\ length' &:: PreList\;a\;Int \to Int \end{aligned}$$

$size$ and $length$ are catamorphisms

$$\begin{aligned} cata &:: Functor\;f \Rightarrow (f\;a \to a) \to Fix\;f \to a \\ cata\;f &= f \circ fmap\;(cata\;f) \circ outFix \\ size &= cata\;size' \\ length &= cata\;length' \end{aligned}$$

Catamorphisms

Slide 12

$$\begin{aligned} cata &:: (f\;a \to a) \to Fix\;f \to a \\ cata\;f &= f \circ fmap\;(cata\;f) \circ outFix \end{aligned}$$

- downward ($\kappa\alpha\tau\alpha$) morphisms
- capture recursion patterns found in (simple) accumulations
- (might) help avoid boilerplate code
- essentially generic versions of $foldr$

$$foldr :: \underbrace{(a \to b \to b) \to b}_{PreList\;a\;b\;\to\;b} \to List\;a \to b$$

- fusion: $h \circ f = f' \circ map\;h \Longrightarrow h \circ cata\;f = cata\;f'$ if h is strict

CATAMORPHISMS – EXAMPLES I

Slide 13

Sums over lists

$$\begin{aligned} sum &= cata\ sum' \\ sum'\ Nil &= 0 \\ sum'\ (Cons\ m\ n) &= m + n \end{aligned}$$

List concatenation

$$\begin{aligned} append\ xs &= cata\ (append'\ xs) \\ append'\ xs\ Nil &= xs \\ append'\ xs\ (Cons\ y\ ys) &= inFix\ (Cons\ y\ ys) \end{aligned}$$

CATAMORPHISMS – EXAMPLES II

Slide 14

List flattening

$$\begin{aligned} concat &= cata\ concat' \\ concat'\ Nil &= Nil \\ concat'\ (Cons\ xs\ ys) &= append\ xs\ ys \end{aligned}$$

Converting Nat to Int

$$\begin{aligned} int &= cata\ int' \\ int'\ Zero &= 0 \\ int'\ (Suc\ n) &= n + 1 \end{aligned}$$

A DIFFERENT RECURSION PATTERN

Slide 15

$$\begin{aligned} nat &:: Int \to Nat \\ nat\ 0 &= inFix\ Zero \\ nat\ n &= inFix\ (Suc\ (nat\ (n-1))) \end{aligned}$$

$$\begin{aligned} nat\ n &= inFix\ (nat'\ n) \\ nat'\ 0 &= fmap\ nat\ Zero \\ nat'\ n &= fmap\ nat\ (Suc\ (n-1)) \end{aligned}$$

$$\begin{aligned} nat\ n &= inFix\ (fmap\ nat\ (nat'\ n)) \\ nat'\ 0 &= Zero \\ nat'\ n &= Suc\ (n-1) \end{aligned}$$

ANAMORPHISMS

Slide 16

$$\begin{aligned} ana &:: (a \to f\ a) \to a \to Fix\ f \\ ana\ f &= inFix \circ fmap\ (ana\ f) \circ f \end{aligned}$$

- ➜ upward ($\alpha\nu\alpha$) morphisms
- ➜ dual to catamorphisms
- ➜ generation (unfolding) of data structures
- ➜ fusion: $f \circ h = fmap\ h \circ f' \implies ana\ f \circ h = ana\ f'$

Anamorphisms – Examples

Slide 17

Replication

$$\begin{aligned} repl &:: a \to Int \to List\ a \\ repl\ x &= ana\ (repl'\ x) \\ repl'\ x\ 0 &= Nil \\ repl'\ x\ n &= Cons\ x\ (n - 1) \end{aligned}$$

Zipping

$$\begin{aligned} zip &:: (List\ a,\ List\ b) \to List\ (a, b) \\ zip\ (xs,\ ys) &= ana\ (zip'\ (xs, ys)) \\ zip'\ (In\ (Cons\ x\ xs),\ In\ (Cons\ y\ ys)) &= Cons\ (x, y)\ (xs, ys) \\ zip'\ \_ &= Nil \end{aligned}$$

Hylomorphisms

Slide 18

$$\begin{aligned} hylo &:: Functor\ f \Rightarrow (f\ a \to a) \to (b \to f\ b) \to b \to a \\ hylo\ f\ g &= cata\ f \circ ana\ g \end{aligned}$$

- ➜ the composition of $cata$ and ana is Turing-complete
- ➜ many functions are natural hylomorphisms
- ➜ different (fused) implementation

$$hyle\ f\ g = f \circ fmap\ (hylo\ f\ g) \circ g$$

So what?

Slide 19

- ➜ morphisms make reasoning about programs easier
- ➜ calculational program construction
- ➜ opportunities for optimisation (fusion)
- ➜ help avoid boilerplate code
- ➜ do we need explicit recursion?
- ➜ need language support
- ➜ not really useful in Haskell

What can we use in Haskell?

Type-safe transformations

Slide 20

$$\begin{aligned} &\textbf{class}\ Typeable\ a \\ &cast :: (Typeable\ a,\ Typeable\ b) \Rightarrow a \to Maybe\ b \end{aligned}$$

- ➜ $cast$ implements the type-safe (runtime) cast operation
- ➜ $cast\ x$ yields $Just\ x$ if a and b are the same type and $Nothing$ otherwise

$$\begin{aligned} mkT &:: (Typeable\ b,\ Typeable\ a) \Rightarrow (b \to b) \to (a \to a) \\ mkT\ f &= \textbf{case}\ cast\ f\ \textbf{of} \\ &\qquad Just\ g \to g \\ &\qquad Nothing \to id \end{aligned}$$

- ➜ means "make a transformation"
- ➜ converts a monomorphic function to a function over arbitrary types
- ➜ the base function is only applied to suitably typed terms

Slide 21

USING GENERIC TRANSFORMATIONS

➜ defining the transformation

$$\begin{aligned}&\mathit{upper} :: \mathit{Typeable}\ a \Rightarrow a \rightarrow a\\&\mathit{upper} = \mathit{mkT}\ \mathit{toUpper}\end{aligned}$$

➜ applying it

$$\begin{aligned}&\mathit{upper}\ \text{'a'} &&= \text{'A'}\\&\mathit{upper}\ \mathit{False} &&= \mathit{False}\\&\mathit{upper}\ \text{"abc"} &&= \text{"abc"}\end{aligned}$$

➜ applying it *everywhere*

$$\begin{aligned}&\mathit{everywhere} :: \mathit{Data}\ a \Rightarrow (\mathit{forall}\ b.\ \mathit{Data}\ b \Rightarrow b \rightarrow b) \rightarrow a \rightarrow a\\&\mathit{everywhere}\ \mathit{upper}\ \text{'a'} = \text{'A'}\\&\mathit{everywhere}\ \mathit{upper}\ \text{"abc"} = \text{"ABC"}\\&\mathit{everywhere}\ \mathit{upper}\ [\text{"abc"}, \text{"cde"}] = [\text{"ABC"}, \text{"CDE"}]\\&\mathit{everywhere}\ \mathit{upper}\ (\mathit{Just}\ \text{"abc"}) = \mathit{Just}\ \text{"ABC"}\end{aligned}$$

Slide 22

HOW DOES IT WORK?

$$\begin{aligned}&\textbf{class}\ \mathit{Typeable}\ a \Rightarrow \mathit{Data}\ a\ \textbf{where}\\&\quad \mathit{gmapT} :: (\mathit{forall}\ b.\ \mathit{Data}\ b \Rightarrow b \rightarrow b) \rightarrow a \rightarrow a\\&\textbf{instance}\ \mathit{Data}\ a \Rightarrow \mathit{Data}\ (\mathit{Maybe}\ a)\ \textbf{where}\\&\quad \mathit{gmapT}\ f\ \mathit{Nothing} = \mathit{Nothing}\\&\quad \mathit{gmapT}\ f\ (\mathit{Just}\ x) = \mathit{Just}\ (f\ x)\\&\textbf{instance}\ \mathit{Data}\ a \Rightarrow \mathit{Data}\ [a]\ \textbf{where}\\&\quad \mathit{gmapT}\ f\ [\,] \qquad = [\,]\\&\quad \mathit{gmapT}\ f\ (x : xs) = f\ x : f\ xs\end{aligned}$$

➜ gmapT applies f to all $\mathit{immediate}$ children of the current node

➜ not recursive

➜ similar to fmap over a pretype

Slide 23

IMPLEMENTATION OF *everywhere*

$$\begin{aligned}&\mathit{everywhere} &&:: \mathit{Data}\ a \Rightarrow (\mathit{forall}\ b.\ \mathit{Data}\ b \Rightarrow b \rightarrow b) \rightarrow a \rightarrow a\\&\mathit{everywhere}\ f\ x &&= f\ (\mathit{gmapT}\ (\mathit{everywhere}\ f)\ x)\end{aligned}$$

➜ higher-order polymorphism

➜ compare this to $\mathit{cata}\ f = f \circ \mathit{fmap}\ (\mathit{cata}\ f) \circ \mathit{outFix}$

$$\begin{aligned}\mathit{everywhere}\ \mathit{upper}\ \text{'a'} &= \mathit{upper}\ (\mathit{gmapT}\ (\mathit{everywhere}\ \mathit{upper})\ \text{'a'})\\&= \mathit{upper}\ \text{'a'} = \text{'A'}\\\mathit{everywhere}\ \mathit{upper}\ (\mathit{Just}\ \text{'a'}) &= \mathit{upper}\ (\mathit{gmapT}\ (\mathit{everywhere}\ \mathit{upper})\ (\mathit{Just}\ \text{'a'}))\\&= \mathit{upper}\ (\mathit{Just}\ (\mathit{upper}\ \text{'a'}))\\&= \mathit{upper}\ (\mathit{Just}\ \text{'A'}) = \mathit{Just}\ \text{'A'}\end{aligned}$$

Slide 24

SUMMARY OF THE APPROACH (SO FAR)

➜ define a (monomorphic) function f which does the work

➜ create a transformation with mkT: applies f to terms of the right type, identity for everything else

➜ gmapT applies the transformation to all immediate children

➜ $\mathit{everywhere}$ applies it everywhere

QUERIES

$$\begin{aligned} &mkQ && :: (Typeable\ a,\ Typeable\ b) \Rightarrow r \rightarrow (b \rightarrow r) \rightarrow (a \rightarrow r) \\ &mkQ\ r\ q\ a = && \mathbf{case}\ cast\ a\ \mathbf{of} \\ & && \quad Just\ b \quad \rightarrow q\ b \\ & && \quad Nothing \rightarrow r \end{aligned}$$

Slide 25

- ➜ mkQ creates queries
- ➜ applies the base function if the argument has the right type; yields the default values otherwise

$$\begin{aligned} everything &:: Data\ a \Rightarrow (r \rightarrow r \rightarrow r) \rightarrow (forall\ b.\ Data\ b \Rightarrow b \rightarrow r) \\ &\rightarrow a \rightarrow r \end{aligned}$$

- ➜ $everything$ folds a tree with the given query

QUERIES – EXAMPLES

- ➜ defining the query

$$\begin{aligned} &int :: Typeable\ a \ {=}{>}\ a {-}{>} Int \\ &int = mkQ\ (0 :: Int)\ id \end{aligned}$$

- ➜ applying it

Slide 26

$$\begin{aligned} int\ (1 :: Int) &= 1 \\ int\ [1 :: Int, 2] &= 0 \end{aligned}$$

- ➜ folding a term

$$\begin{aligned} everything\ (+)\ int\ (1 :: Int) &= 1 \\ everything\ (+)\ int\ (Just\ 1) &= 1 \\ everything\ (+)\ int\ [1,2,3] &= 6 \\ everything\ (+)\ int\ [[1,2],[3,4]] &= 10 \end{aligned}$$

QUERIES – IMPLEMENTATION

$$\begin{aligned} &\mathbf{class}\ Typeable\ a \Rightarrow Data\ a\ \mathbf{where} \\ &\quad gmapQ :: (forall\ b.\ Data\ b \Rightarrow b \rightarrow r) \rightarrow a \rightarrow [r] \\ &\mathbf{instance}\ Data\ a \Rightarrow Data\ (Maybe\ a)\ \mathbf{where} \\ &\quad gmapQ\ f\ Nothing = [] \\ &\quad gmapQ\ f\ (Just\ x) = [f\ x] \\ &\mathbf{instance}\ Data\ a \Rightarrow Data\ [a]\ \mathbf{where} \\ &\quad gmapQ\ f\ [] \qquad = [] \\ &\quad gmapQ\ f\ (x : xs) = [f\ x,\ f\ xs] \end{aligned}$$

Slide 27

- ➜ applies f to all $immediate$ children of the current node
- ➜ not recursive

IMPLEMENTATION OF $everything$

$$\begin{aligned} &everything && :: Data\ a \Rightarrow (r \rightarrow r \rightarrow r) \\ & && \rightarrow (forall\ b.\ Data\ b \Rightarrow b \rightarrow r) \\ & && \rightarrow a \rightarrow r \\ &everything\ op\ f\ x = && foldl\ op\ (f\ x)\ (gmapQ\ (everything\ op\ f)\ x) \end{aligned}$$

Slide 28

- ➜ $gmapQ$ yields a list of results
- ➜ $f\ x$ computes the result for the current node
- ➜ $foldl$ combines them using the provided operator

Slide 29

That's really verbose!

$$\mathbf{data}\ \mathit{Tree}\ a\ = \mathit{Empty} \mid \mathit{Tree}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$
$$\mathbf{deriving}(\mathit{Typeable},\ \mathit{Data})$$

$$\mathit{everywhere}\ \mathit{upper}\ (\mathit{Node}\ \text{“abc”}\ (\mathit{Node}\ \text{“d”}\ \mathit{Empty}\ \mathit{Empty})\ \mathit{Empty})$$
$$=\ \mathit{Node}\ \text{“ABC”}\ (\mathit{Node}\ \text{“D”}\ \mathit{Empty}\ \mathit{Empty})\ \mathit{Empty}$$
$$\mathit{everything}\ (+)\ \mathit{int}\ (\mathit{Node}\ 1\ (\mathit{Node}\ 2\ \mathit{Empty}\ \mathit{Empty})\ \mathit{Empty})\ =\ 3$$

➜ you need a fairly recent version of GHC
➜ import Data.Generics

Slide 30

Summary

➜ avoiding explicit recursion is a good idea (cf. map)
➜ less boilerplate code, less opportunities for mistakes
➜ reasoning about programs is easier
➜ looks nice too

Design patterns for functional programming

Slide 31

References

➜ *Functional programming with bananas, lenses, envelopes and barbed wire.* Erik Meijer, Maarten Fokkinga, and Ross Paterson. In “Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture”, pages 124–144, 1991. `http://portal.acm.org/citation.cfm?id=128035&dl=ACM&coll=portal`

➜ *Scrap your boilerplate: a practical approach to generic programming.* Ralf Lämmel and Simon Peyton Jones. In “Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)”, 2003. `http://research.microsoft.com/Users/simonpj/Papers/hmap/index.htm`