Everything is a list arrow

Functional programming at typLAB

Erik Hesselink, typLAB

April 24, 2010

Who are we?

typLAB: started in June 2009.

Silk: semantic text editor on the web.

http://blog.typlab.com
http://silkapp.com

silk app

Who are we?

- Four people.
- Commercial web development experience.
- Academic software technology background.
- Love the web, Haskell and Javascript.
- No customers.

What is this about arrows?

- Our editor stores documents as XML.
- Our user interface is dynamically built using HTML.
- There is a pattern to the techniques we use for both:

What is this about arrows?

- Our editor stores documents as XML.
- Our user interface is dynamically built using HTML.
- There is a pattern to the techniques we use for both:

List arrows

What is an arrow?

An arrow is:

- An abstraction over functions.
- Something with an input and an output.
- A thing that can be composed.

What is an arrow?

An arrow is:

- An abstraction over functions.
- Something with an input and an output.
- A thing that can be composed.

Actually, that's a Category!

Category class

$$\begin{aligned}&\textbf{class}\ \mathit{Category}\ (\rightsquigarrow)\ \textbf{where}\\ &\quad \mathit{id}\ \ :: a \rightsquigarrow a\\ &\quad (\circ) :: (b \rightsquigarrow c) \longrightarrow (a \rightsquigarrow b) \longrightarrow (a \rightsquigarrow c)\end{aligned}$$

So what do we add to make it an arrow?

- Lifting normal functions into the arrow.
- Passing through values unchanged.

Arrow class

$$\textbf{class}\ \mathit{Category}\ (\rightsquigarrow) \Rightarrow \mathit{Arrow}\ (\rightsquigarrow)\ \textbf{where}$$
$$\quad \mathit{arr}\ \ :: (a \to b) \to (a \rightsquigarrow b)$$
$$\quad \mathit{first} :: (a \rightsquigarrow b) \to ((a, c) \rightsquigarrow (b, c))$$

What instances do we have by default?

- Functions, obviously.
- Monads, too! Or rather, their Kleisli arrow:

$$\textbf{newtype}\ \mathit{Kleisli}\ m\ a\ b = \mathit{Kleisli}\ (a \to m\ b)$$

The Document Object Model (DOM)

The Document Object Model (DOM) is an object representation for XML and HTML trees.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Node} = \mathit{Attribute}\ \mathit{Name}\ \mathit{Value} \\
&\qquad\qquad \mid\ \mathit{Node}\ \mathit{Name}\ [\mathit{Node}]\ [\mathit{Node}] \\
&\qquad\qquad \mid\ \mathit{Text}\ \mathit{Value} \\
&\textbf{type}\ \mathit{Name} = \mathit{String} \\
&\textbf{type}\ \mathit{Value} = \mathit{String}
\end{aligned}
$$

DOM - example

Consider this snippet of HTML:

```
<ul id="numbers">
  <li>One</li>
  <li>Two</li>
</ul>
```

We can represent it in Haskell as:

```
Node "ul"
     [Attribute "id" "numbers"]
     [Node "li" [] [Text "One"]
     ,Node "li" [] [Text "Two"]
     ]
```

Functions on DOM Nodes

Functions on *Node*s can naturally be represented as functions $Node \longrightarrow [Node]$.

Retrieving child nodes:

$$getChildren :: Node \longrightarrow [Node]$$

Filtering, representing failure by $[]$.

$$isA :: (Node \longrightarrow Bool) \longrightarrow Node \longrightarrow [Node]$$

Concatenation, choice, element creation, ...

List arrows

The functions $Node \longrightarrow [Node]$ are list arrows (if generalized).

$$\begin{aligned}
&\textbf{data } \mathit{ListArrow}\ a\ b = \mathit{LA}\ (a \longrightarrow [b]) \\
&\textbf{instance } \mathit{Category}\ \mathit{ListArrow}\ \textbf{where} \\
&\quad \mathit{id} = \mathit{LA}\ (\lambda x \longrightarrow [x]) \\
&\quad (\mathit{LA}\ g) \circ (\mathit{LA}\ f) = \mathit{LA}\ (\lambda x \longrightarrow [z \mid y \leftarrow f\ x, z \leftarrow g\ y]) \\
&\textbf{instance } \mathit{Arrow}\ \mathit{ListArrow}\ \textbf{where} \\
&\quad \mathit{arr}\ \ f = \mathit{LA}\ (\lambda x \longrightarrow [f\ x]) \\
&\quad \mathit{first}\ (\mathit{LA}\ f) = \mathit{LA}\ (\lambda(x, y) \longrightarrow [(z, y) \mid z \leftarrow f\ x])
\end{aligned}$$

List arrow functions

We can define basic arrows:

```
getChildren :: ListArrow Node Node
getChildren = LA $ λnode →
  case node of
    (Node _ _ cs) → cs
    _             → []
isA :: (a → Bool) → ListArrow a a
isA pred = LA $ λx →
  if   pred x
  then [x]
  else []
```

ArrowPlus

We can also define concatenation on these arrows:

$$\textbf{class } \mathit{ArrowPlus}\ (\rightsquigarrow)\ \textbf{where}$$
$$\quad (\oplus) :: (a \rightsquigarrow b) \to (a \rightsquigarrow b) \to (a \rightsquigarrow b)$$
$$\textbf{instance } \mathit{ArrowPlus}\ \mathit{ListArrow}\ \textbf{where}$$
$$\quad (\mathit{LA}\ f) \oplus (\mathit{LA}\ g) = \mathit{LA}\ (\lambda x \to f\ x +\!\!+ g\ x)$$

Now we can apply an arrow everywhere in a tree, and gather the results.

$$\mathit{deep} :: \mathit{ListArrow}\ \mathit{Node}\ \mathit{Node} \to \mathit{ListArrow}\ \mathit{Node}\ \mathit{Node}$$
$$\mathit{deep}\ f = f \oplus (\mathit{deep}\ f \circ \mathit{getChildren})$$

Building a table of contents

These building blocks now allow you to gather all headers in a document, to build a table of contents:

$$
\begin{aligned}
&\mathit{hasName} :: \mathit{String} \to \mathit{ListArrow}\ \mathit{Node}\ \mathit{Node} \\
&\mathit{hasName}\ \mathit{name} = \mathit{isA}\ \$\ \lambda(\mathit{Node}\ n\ \_\ \_) \to n \equiv \mathit{name} \\
&\mathit{isH} :: \mathit{ListArrow}\ \mathit{Node}\ \mathit{Node} \\
&\mathit{isH} = \mathit{foldl1}\ (\oplus)\ (\mathit{map}\ \mathit{hasName}\ [h1\,..\,h6]) \\
&\mathit{allHeaders} :: \mathit{ListArrow}\ \mathit{Node}\ \mathit{Node} \\
&\mathit{allHeaders} = \mathit{deep}\ \mathit{isH}
\end{aligned}
$$

List arrows are everythere

You can recognise this kind of structure in a lot of places:

- XPath, a language for querying XML documents.

XPath	Haskell
/ul/li	*hasName* `"li"` ∘ *getChildren* ∘ *hasName* `"ul"`
//div	*deep* (*hasName* `"div"`)
//b | //strong	*deep* (*hasName* `"b"`) ⊕ *deep* (*hasName* `"strong"`)

List arrows are everythere

You can recognise this kind of structure in a lot of places:

- jQuery, a Javascript library for cross-browser DOM manipulation:

jQuery	Haskell
$().is("ul").children().is("li")	*hasName* `"li"` $\circ$ *getChildren* $\circ$ *hasName* `"ul"`
$("div")	*deep* (*hasName* `"div"`)
$("b").add("strong")	*deep* (*hasName* `"b"`) $\oplus$ *deep* (*hasName* `"strong"`)

List arrows are everythere

You can recognise this kind of structure in a lot of places:

- jQuery, a Javascript library for cross-browser DOM manipulation:

jQuery	Haskell
$().is("ul").children().is("li")	*hasName* `"li"` $\circ$ *getChildren* $\circ$ *hasName* `"ul"`
$("div")	*deep* (*hasName* `"div"`)
$("b").add("strong")	*deep* (*hasName* `"b"`) $\oplus$ *deep* (*hasName* `"strong"`)

Actually, jQuery is more like the list *monad*.

List arrows at typLAB

At typLAB, we use this in a lot of places:

- We use an XML database to store document, which we query using XPath.
- We use jQuery on the client side.
- We use the HXT library in our Haskell server code to manipulate and create XML.
- We've written an arrow-based library in Javascript to build *reactive* user interfaces.

Problems with list arrows

We've encountered a couple of issues with list arrows:

- They can be difficult to grasp and restrictive to program with.
  - ... but you gain structure.
- They can generate unexpected Cartesian products of lists.
- Large amounts of intermediate lists can be generated.
  - ... you need a good compiler.
- Performing side-effects inside an arrow can be problematic.
  - ... Haskell has a type system, Javascript doesn't.

Questions?

Reactive lists in Javascript

- Create list like objects.
- Declare relations between them.
- Changing some updates the others.

```
var input = new List([1,2]);
var output = input.map(
    function (x) { return x * 2; }
  );
output.list; // [2,4]
input.push(3);
output.list; // [2,4,6]
```

Reactive nodes in Javascript

- Wrap DOM nodes in an object.
- Give them a reactive list of child nodes.
- Manipulate these using list arrows.

```
var isHeader = sum(hasName("h1"), hasName("h2"));
var toc = mkElem("ul")(
            seq( deep(isHeader)
               , mkElem("li")(getText())
               )
            );
var rBody = new Node(document.body);
var tocOutput = toc.run(rBody);
```