Trees as fixed point types

Balázs Kőműves

December 2011

Some Haskell syntax in 2 minutes

Some syntax reminder:

-- this is a comment	
$add :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$	-- type annotation
$add\ x\ y = x + y$	-- function definition
$add_1 :: \mathsf{Num}\ a \Rightarrow a \to a \to a$	-- type class constraint
$add_1 = \backslash x\ y \mapsto x + y$	-- anonyomous function
$addThree = add\ 3$	-- partial application
$(f \circ g)\ x = f\ (g\ (x))$	-- function composition
$f\ \$\ g\ \$\ x\ = f\ (g\ (x))$	-- unix-style "pipes"
$const :: a \to b \to a$	
$const\ x\ \_ = x$	-- wildcard

Trees

A binary tree with numbers at the leaves:

$$\begin{aligned}\textbf{data}\ \mathsf{Tree} &= \mathsf{Leaf}\quad \mathsf{Int}\\ &\mid\ \mathsf{Branch}\ \mathsf{Tree}\ \mathsf{Tree}\end{aligned}$$

A binary tree with parametrized by the data type at the leaves:

$$\begin{aligned}\textbf{data}\ \mathsf{Tree}\ a &= \mathsf{Leaf}\ a\\ &\mid\ \mathsf{Branch}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)\end{aligned}$$

As an example, here is a "flattening" function:

$$\begin{aligned}&\mathit{flatten} :: \mathsf{Tree}\ a \to [a]\\ &\mathit{flatten}\ (\mathsf{Leaf}\ x) &&= [x]\\ &\mathit{flatten}\ (\mathsf{Branch}\ \mathit{left}\ \mathit{right}) &&= \mathit{flatten}\ \mathit{left} + \!\!\!+\ \mathit{flatten}\ \mathit{right}\end{aligned}$$

Expression trees

Expressions can be conveniently encoded by trees:

```
data Expr = Kst Int                -- constant
          | Var String             -- variable
          | Add Expr Expr          -- sum
          | Mul Expr Expr          -- product
          | Fun String [Expr]      -- function call
```

The following example encodes $x + 5\sin(y)$:

$$ex = \mathsf{Add}\ (\mathsf{Var}\ \texttt{"x"})\ (\mathsf{Mul}\ (\mathsf{Kst}\ 5)\ (\mathsf{Fun}\ \texttt{"sin"}\ [\mathsf{Var}\ \texttt{"y"}]))$$

We often want to manipulate such trees. For example we may want to simplify Add (Kst 0) e to e, and so on. Very similar tasks appear in compilers, and many other contexts.

Annotations

Another frequent task is to compute some value for each node (or subtree), and remember them. Examples are the size of a subtree, the hash of a subtree, the type of a subexpression. We could change our type to incorporate these extra data:

$$
\begin{aligned}
\textbf{data}\ \mathsf{HashedExpr} = {} & \mathsf{Kst}\ \ \mathsf{Hash}\ \mathsf{Int} \\
\mid {} & \mathsf{Var}\ \ \mathsf{Hash}\ \mathsf{String} \\
\mid {} & \mathsf{Add}\ \mathsf{Hash}\ \mathsf{Expr}\ \mathsf{Expr} \\
\mid {} & \mathsf{Mul}\ \mathsf{Hash}\ \mathsf{Expr}\ \mathsf{Expr} \\
\mid {} & \mathsf{Fun}\ \ \mathsf{Hash}\ \mathsf{String}\ [\mathsf{Expr}]
\end{aligned}
$$

But that looks very inconvenient: We have to rewrite all the code, convert between very similar-looking types, etc. It involves a lot of boring "boilerplate" code.

Trees as fixed-point types

We can describe the shape of a tree by a non-recursive data type:

$$\begin{array}{llll} \textbf{data}\ \mathsf{Expr_1}\ e = & \mathsf{Kst\ Int} & & \text{-- constant} \\ \mid & \mathsf{Var\ String} & & \text{-- variable} \\ \mid & \mathsf{Add}\ e\ e & & \text{-- sum} \\ \mid & \mathsf{Mul}\ e\ e & & \text{-- product} \\ \mid & \mathsf{Fun\ String}\ [e] & & \text{-- function call} \end{array}$$

What happend is that we replace all occurences of Expr on the RHS by the type parameter e. Then we can get back our tree type using a fixed-point operator:

$$\textbf{newtype}\ \mathsf{Mu}\ f = \mathsf{Fix}\ (f\ (\mathsf{Mu}\ f))$$

Now, $\mathsf{Mu\ Expr_1}$ is isomorphic to our original tree type Expr.

Functors

Expr_1 takes a type parameter, and produces a new type. It also does this in a pretty regular way; for example, it is a **functor**. A functor is something which implements the following interface (and satisfies some law):

$$\textbf{class}\ \mathsf{Functor}\ t\ \textbf{where}\ \mathit{fmap} :: (a \to b) \to t\ a \to t\ b$$

In our case, there is only one natural implementation of this:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Functor}\ \mathsf{Expr}_1\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathsf{Kst}\ i \qquad) = \mathsf{Kst}\ i \\
\quad \mathit{fmap}\ f\ (\mathsf{Var}\ v \qquad) = \mathsf{Var}\ v \\
\quad \mathit{fmap}\ f\ (\mathsf{Add}\ x\ y\ ) = \mathsf{Add}\ (f\ x)\ (f\ y) \\
\quad \mathit{fmap}\ f\ (\mathsf{Mul}\ x\ y\ ) = \mathsf{Mul}\ (f\ x)\ (f\ y) \\
\quad \mathit{fmap}\ f\ (\mathsf{Fun}\ n\ \mathit{xs}) = \mathsf{Fun}\ n\ (\mathit{map}\ f\ \mathit{xs})
\end{array}
$$

We can see that this is very mechanical; indeed, the compiler can do this for us.

Annotations with fixed points

With fixed-point types, we can simply "slice in" annotations:

$$\textbf{data}\ \mathsf{Ann}\ f\ a\ b = \mathsf{Ann}\ a\ (f\ b)$$

Here f is the functor, a is the type of the annotations, and b is the type of children, which will be the annotated tree itself. Now we can define an annotated tree simply as

$$\textbf{type}\ \mathsf{Attr}\ a = \mathsf{Mu}\ (\mathsf{Ann}\ \mathsf{Expr}_1\ a)$$

Note that if f implements $\mathsf{Functor}$, then $\mathsf{Ann}\ f\ a$ also does:

$$\begin{array}{l}\textbf{instance}\ \mathsf{Functor}\ t \Rightarrow \mathsf{Functor}\ (\mathsf{Ann}\ t\ a)\ \textbf{where} \\ \quad \mathit{fmap}\ f\ (\mathsf{Ann}\ x\ y) = \mathsf{Ann}\ x\ (\mathit{fmap}\ f\ y)\end{array}$$

Bottom-up transformations I.

While all this is rather simple, it is already quite powerful: For example we can define a generic recursive bottom-up transformation with it. The idea is that since this type of transformation occurs very often, thus we should abstract it away:

$$
\begin{aligned}
&\mathit{transform} :: \mathsf{Functor}\ t \Rightarrow (\mathsf{Mu}\ t \to \mathsf{Mu}\ t) \to (\mathsf{Mu}\ t \to \mathsf{Mu}\ t)\\
&\mathit{transform}\ f = f \circ \mathsf{Fix} \circ \mathit{fmap}\ (\mathit{transform}\ f) \circ \mathit{unFix}
\end{aligned}
$$

where we used a small utility function

$$
\begin{aligned}
&\mathit{unFix} :: \mathsf{Mu}\ t \to t\ (\mathsf{Mu}\ t)\\
&\mathit{unFix}\ (\mathsf{Fix}\ x) = x
\end{aligned}
$$

Bottom-up transformations II.

We use the bottom-up transformation to eliminate additions of zero, and multiplications by zero or one:

$$
\begin{aligned}
&\mathit{elim} :: \mathsf{Mu}\ \mathsf{Expr}_1 \rightarrow \mathsf{Mu}\ \mathsf{Expr}_1 \\
&\mathit{elim}\ (\mathsf{Fix}\ (\mathsf{Add}\ (\mathsf{Fix}\ (\mathsf{Kst}\ 0))\ x)) = x \\
&\mathit{elim}\ (\mathsf{Fix}\ (\mathsf{Mul}\ (\mathsf{Fix}\ (\mathsf{Kst}\ 0))\ x)) = \mathsf{Fix}\ (\mathsf{Kst}\ 0) \\
&\ldots \\
&\mathit{elim}\ \mathit{anythingElse} \qquad\qquad\qquad\quad = \mathit{anythingElse}
\end{aligned}
$$

This is unfortunately a bit ugly because of the annoying Fix noise (but there are solutions for that). Then we can use it:

$$
\begin{aligned}
&\mathit{simplify} :: \mathsf{Mu}\ \mathsf{Expr}_1 \rightarrow \mathsf{Mu}\ \mathsf{Expr}_1 \\
&\mathit{simplify} = \mathit{transform}\ \mathit{elim}
\end{aligned}
$$

Syntax sugar with view patterns

$$
\begin{array}{l}
\textbf{type}\ \mathsf{E} = \mathsf{Mu}\ \mathsf{Expr}_1 \\[1ex]
zero :: \mathsf{E} \to \mathsf{Bool} \\
zero\ (\mathsf{Fix}\ (\mathsf{Kst}\ 0)) = \mathsf{True} \\
zero\ \_ \qquad\qquad\qquad\ \ = \mathsf{False} \\[1ex]
plus :: \mathsf{E} \to \mathsf{Maybe}\ (\mathsf{E}, \mathsf{E}) \\
plus\ (\mathsf{Fix}\ (\mathsf{Add}\ x\ y)) = \mathsf{Just}\ (x, y) \\
plus\ \_ \qquad\qquad\qquad\quad\ \ = \mathsf{Nothing} \\[1ex]
kst :: \mathsf{Int} \to \mathsf{E} \\
kst = \mathsf{Fix} \circ \mathsf{Kst} \\[1ex]
elim :: \mathsf{E} \to \mathsf{E} \\
elim\ (plus \ \ \to \mathsf{Just}\ (x, y))\ |\ zero\ x \qquad\qquad\quad = y \\
elim\ (plus \ \ \to \mathsf{Just}\ (x, y))\ | \qquad\qquad\ zero\ y = x \\
elim\ (times \to \mathsf{Just}\ (x, y))\ |\ zero\ x \lor zero\ y = kst\ 0 \\
elim\ any \qquad\qquad\qquad\qquad\qquad\qquad\qquad\ = any
\end{array}
$$

Foldable

Expr_1 also implements some other interfaces, for example we can **fold** over it:

$$\begin{aligned}
&\textbf{class}\ \mathsf{Foldable}\ f\ \textbf{where}\\
&\quad \mathit{foldl} :: (a \to b \to a) \to a \to f\ b \to a\\
&\quad \mathit{foldr} :: (b \to a \to a) \to a \to f\ b \to a
\end{aligned}$$

Fold is simply a generalization of sum and product for lists:

$$\begin{aligned}
\mathit{sum}\ xs &= \mathit{foldl}\ (+)\ 0\ xs\\
\mathit{prod}\ xs &= \mathit{foldl}\ (*)\ 1\ xs
\end{aligned}$$

One use of $\mathsf{Foldable}$ is to flatten a structure to a list:

$$\begin{aligned}
&\mathit{toList} :: \mathsf{Functor}\ t \Rightarrow t\ a \to [a]\\
&\mathit{toList}\ xs = \mathit{foldr}\ (:)\ [\,]\ xs
\end{aligned}$$

Substructures

Using fold, we can implement the function returning the direct descendants subtrees of the root node:

$$children :: \mathsf{Foldable}\ t \Rightarrow \mathsf{Mu}\ t \rightarrow [\mathsf{Mu}\ t]$$
$$children = toList \circ unFix$$

and also the function returning **all** subtrees:

$$universe :: \mathsf{Foldable}\ t \Rightarrow \mathsf{Mu}\ t \rightarrow [\mathsf{Mu}\ t]$$
$$universe\ x = x : concatMap\ universe\ (children\ x)$$

This is very useful. Suppose for example we want a list of all variables names occuring in an expression. We can do that using *universe* and the list comprehension syntax:

$$variables :: \mathsf{Mu}\ \mathsf{Expr}_1 \rightarrow [\mathsf{String}]$$
$$variables\ expr = [s \mid \mathsf{Fix}\ (\mathsf{Var}\ s) \leftarrow universe\ expr]$$

Often we want to repeat transformations until a normal form is achieved. This can be also abstracted away:

$$\begin{aligned} &\mathit{rewrite} :: \mathsf{Functor}\ t \\ &\qquad\qquad \Rightarrow (\mathsf{Mu}\ t \rightarrow \mathsf{Maybe}\ (\mathsf{Mu}\ t)) \rightarrow (\mathsf{Mu}\ t \rightarrow \mathsf{Mu}\ t) \\ &\mathit{rewrite}\ f = \mathit{transform}\ (\backslash x \mapsto \mathit{maybe}\ x\ (\mathit{rewrite}\ f)\ (f\ \ x)) \end{aligned}$$

There are also versions of *transform* and *rewrite* which maintain a state, or print something to the screen, or do something else which needs a linear order. This needs a third abstraction on the top of Functor and Foldable, called Traversable.

The Traversable class is the following interface:

$$\textbf{class}\ \mathsf{Traversable}\ t\ \textbf{where}$$
$$\quad mapM :: \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)$$

This basically means that we can go through the structure, have some side effects, and collect the results in a structure of the same shape. In particular, we can thread a state through the traversal:

$$mapAccumL$$
$$\quad :: \mathsf{Traversable}\ t$$
$$\quad \Rightarrow (a \to b \to (a, c)) \to a \to t\ b \to (a, t\ c)$$

Bottom-up computing

Suppose we want to compute the size of a tree. This falls into the category of computations when we use already computed values of the subtrees to compute the value for the root. Computer scientist gave the scary name "catamorphism" to this very simple concept :)

$$
\begin{aligned}
&\mathit{cata} :: \mathsf{Functor}\ t \Rightarrow (t\ a \to a) \to \mathsf{Mu}\ t \to a \\
&\mathit{cata}\ f = f \circ \mathit{fmap}\ (\mathit{cata}\ f) \circ \mathit{unFix}
\end{aligned}
$$

We can then use this generic version to compute the size of a tree:

$$
\begin{aligned}
&\mathit{size} :: (\mathsf{Functor}\ t, \mathsf{Foldable}\ t) \Rightarrow \mathsf{Mu}\ t \to \mathsf{Int} \\
&\mathit{size} = \mathit{cata}\ (\backslash x \mapsto 1 + \mathit{sum}\ (\mathit{toList}\ x))
\end{aligned}
$$

Actually *toList* is redundant here; the generalized *sum* works for any type implementing Foldable.

Synthetizing attributes

Most probably we also want the sizes of all subtrees, not only the big tree: We want to **annotate** the nodes with the sizes of the given subtree.

$$\begin{aligned}
&\mathit{synthetize} :: \mathsf{Functor}\ t \Rightarrow (t\ a \to a) \to \mathsf{Mu}\ t \to \mathsf{Attr}\ t\ a \\
&\mathit{synthetize}\ f\ x = \mathsf{Fix}\ (\mathsf{Ann}\ (f\ \mathit{as})\ y)\ \mathbf{where} \\
&\quad y\ \ = \mathit{fmap}\ (\mathit{synthetize}\ f)\ (\mathit{unFix}\ x) \\
&\quad \mathit{as} = \mathit{fmap}\ (\backslash(\mathsf{Fix}\ (\mathsf{Ann}\ a\ _)) \mapsto a)\ y
\end{aligned}$$

We can also do top-down computations (for example to compute the **depths** of nodes), which is even simpler:

$$\begin{aligned}
&\mathit{inherit} :: \mathsf{Functor}\ t \Rightarrow (a \to a) \to a \to \mathsf{Mu}\ t \to \mathsf{Attr}\ t\ a \\
&\mathit{inherit}\ f\ a\ (\mathsf{Fix}\ x) = \mathsf{Fix}\ (\mathsf{Ann}\ b\ y)\ \mathbf{where} \\
&\quad b = f\ a \\
&\quad y = \mathit{fmap}\ (\mathit{inherit}\ f\ b)\ x
\end{aligned}$$

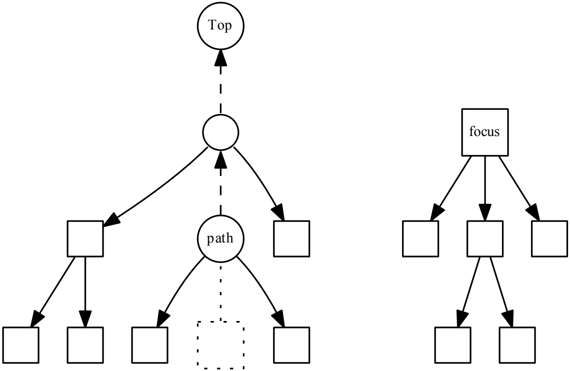
Zipper I.

The zipper is an inmutable data structure which encodes a tree together with a position (called focus) in a tree, and allows moving the position and making local changes. We can implement a version of it generically, so it will work for any tree!

$$
\begin{aligned}
\textbf{data}\ \mathsf{Path}\ t &= \mathsf{Top} \\
&\mid\ \mathsf{Path}\ (t\ (\mathsf{Either}\ (\mathsf{Mu}\ t)\ (\mathsf{Path}\ t))) \\
\textbf{data}\ \mathsf{Loc}\ t &= \mathsf{Loc}\ \{\mathit{focus} :: \mathsf{Mu}\ t, \mathit{path} :: \mathsf{Path}\ t\}
\end{aligned}
$$

This is best explained by pictures.

Zipper II.



The $\mathsf{Loc}\ t$ data type. Boxes are of type $\mathsf{Mu}\ t$, while circles are $\mathsf{Path}\ t$. The dashed arrows are the Right children. The dotted box denotes the place where the subtree at the focus (shown separately on the right) would fit in.

Mathematically satisfying constructions

We can annotate the nodes of a tree by zippers focused at that particular node.

$$locations :: \mathsf{Traversable}\ t \Rightarrow \mathsf{Mu}\ t \to \mathsf{Attr}\ t\ (\mathsf{Loc}\ t)$$

Since we can modify the subtree at the focus, as a consequence (actually this has a simpler implementation, too), we can also annotate the nodes by functions replacing (or changing) that subtree:

$$contexts :: \mathsf{Traversable}\ t \Rightarrow \mathsf{Mu}\ t \to \mathsf{Attr}\ t\ (\mathsf{Mu}\ t \to \mathsf{Mu}\ t)$$

This can be useful for mutation testing; say we want to perturb all the constants in our expression, one at a time: This can be easily done using *contexts*.

Mutually recursive types

The above works for a single tree. Sometimes we want to interleave more types of trees; for example an imperative language can contain statements and expression, which are mixed together:

$$\begin{aligned}
&\textbf{data}\ \mathsf{Stmt} &= &\ \mathsf{Block}\ [\mathsf{Stmt}] \\
& &| &\ \mathsf{Assign}\ \mathsf{String}\ \mathsf{Expr} \\
&\textbf{data}\ \mathsf{Expr} &= &\ \mathsf{Kst}\ \mathsf{Int} \\
& &| &\ \mathsf{Var}\ \mathsf{String} \\
& &| &\ \mathsf{Add}\ \mathsf{Expr}\ \mathsf{Expr} \\
& &| &\ \mathsf{Fun}\ \mathsf{String}\ [\mathsf{Expr}] \\
& &| &\ \mathsf{Do}\ [\mathsf{Stmt}]\ \mathsf{Expr}
\end{aligned}$$

The above framework does not work in this setting. However, it should be possible to generalize.

Bifunctors

We fully "open up" both tree types:

$$
\begin{aligned}
\textbf{data}\ \mathsf{Stmt}_1\ s\ e &= \mathsf{Block}\ [\,s\,] \\
&\mid\ \mathsf{Assign}\ \mathsf{String}\ e \\
\textbf{data}\ \mathsf{Expr}_1\ s\ e &= \mathsf{Kst}\ \mathsf{Int} \\
&\mid\ \mathsf{Var}\ \mathsf{String} \\
&\mid\ \mathsf{Add}\ e\ e \\
&\mid\ \mathsf{Fun}\ \mathsf{String}\ [\,e\,] \\
&\mid\ \mathsf{Do}\ [\,s\,]\ e
\end{aligned}
$$

These become **bifunctors**: They have two arguments and are functors in both.

Bifunctors and fixed point types

We have two fixed point types now, corresponding to the two types Stmt and Expr. We call them left and right (in this example, Stmt is the left one):

$$\begin{aligned}&\textbf{newtype}\ \mathsf{Mu_L}\ f\ g = \mathsf{Fix_L}\ \{unFix_L :: f\ (\mathsf{Mu_L}\ f\ g)\ (\mathsf{Mu_R}\ f\ g)\}\\ &\textbf{newtype}\ \mathsf{Mu_R}\ f\ g = \mathsf{Fix_R}\ \{unFix_R :: g\ (\mathsf{Mu_L}\ f\ g)\ (\mathsf{Mu_R}\ f\ g)\}\end{aligned}$$

Now we can recover the interleaved trees:

$$\begin{aligned}&\textbf{type}\ \mathsf{S} = \mathsf{Mu_L}\ \mathsf{Stmt_1}\ \mathsf{Expr_1}\\ &\textbf{type}\ \mathsf{E} = \mathsf{Mu_R}\ \mathsf{Stmt_1}\ \mathsf{Expr_1}\end{aligned}$$

Bifunctor class and *transformBi*

We can define a BiFunctor class:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{BiFunctor}\ f\ \textbf{where} \\
\quad \mathit{fmapLeft}\ \ :: (a \to b) \to f\ \ a\ \ c \to f\ \ b\ \ c \\
\quad \mathit{fmapRight} :: (b \to c) \to f\ \ a\ \ b \to f\ \ a\ \ c \\
\quad \mathit{fmapBoth}\ \ :: (a \to c) \to (b \to d) \to f\ \ a\ \ b \to f\ \ c\ \ d
\end{array}
$$

And use to the define generic bottom-up transformations:

$$
\begin{array}{l}
\mathit{transformBiL} :: (\mathsf{BiFunctor}\ f, \mathsf{BiFunctor}\ g) \\
\qquad\qquad\qquad \Rightarrow (\mathsf{Mu_L}\ f\ \ g \to \mathsf{Mu_L}\ f\ \ g) \to (\mathsf{Mu_R}\ f\ \ g \to \mathsf{Mu_R}\ f\ \ g) \\
\qquad\qquad\qquad \to\ \ \mathsf{Mu_L}\ f\ \ g \to \mathsf{Mu_L}\ f\ \ g \\
\mathit{transformBiL}\ u\ \ v = \mathit{goL}\ \textbf{where} \\
\quad \mathit{goL} = u \circ \mathsf{Fix_L}\ \circ \mathit{fmapBoth}\ \mathit{goL}\ \mathit{goR} \circ \mathit{unFix}_L \\
\quad \mathit{goR} = v \circ \mathsf{Fix_R}\ \circ \mathit{fmapBoth}\ \mathit{goL}\ \mathit{goR} \circ \mathit{unFix}_R
\end{array}
$$

In our case, the type specalizes to $(\mathsf{S} \to \mathsf{S}) \to (\mathsf{E} \to \mathsf{E}) \to \mathsf{S} \to \mathsf{S}$. There is another one for expressions, with the type $(\mathsf{S} \to \mathsf{S}) \to (\mathsf{E} \to \mathsf{E}) \to \mathsf{E} \to \mathsf{E}$.