Inside 206-105

Existential Pontification and Generalized Abstract Digressions

Bananas, Lenses, Envelopes and Barbed Wire A Translation Guide

One of the papers I've been slowly rereading since summer began is "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire", by Erik Meijer, Maarten Fokkinga and Ross Paterson. If you want to know what {cata,ana,hylo,para}morphisms are, this is the paper to read: section 2 gives a highly readable formulation of these morphisms for the beloved linked list.

Last time, however, my eyes got a little bit glassy when they started discussing algebraic data types, despite having used and defined them in Haskell; part of me felt inundated in a sea of triangles, circles and squiggles, and by the time they reached the laws for the basic combinators, I might as well have said, "It's all math to me!"

A closer reading revealed that, actually, all of these algebraic operators can be written out in plain Haskell, and for someone who has been working with Haskell for a little bit of time, this can provide a smoother (albeit more verbose) reading. Thus, I present this translation guide.

Type operators. By convention, types are $A, B, C \ldots$ on the left and `a, b, c...` on the right. We distinguish these from function operators, though the paper does not and relies on convention to distinguish between the two.

$A \dagger B \Leftrightarrow$ `Bifunctor t => t a b`
$A_F \Leftrightarrow$ `Functor f => f a`
$A* \Leftrightarrow$ `[a]`
$D \parallel D' \Leftrightarrow$ `(d, d')`
$D \mid D' \Leftrightarrow$ `Either d d'`
$_I \Leftrightarrow$ `Identity`
$\underline{D} \Leftrightarrow$ `Const d`
$A_{(FG)} \Leftrightarrow$ `(Functor f, Functor g) => g (f a)`
$A_{(F \dagger G)} \Leftrightarrow$ `(Bifunctor t, Functor f, Functor g) => Lift t f g a`
$\mathbf{1} \Leftrightarrow$ `()`

(For the pedantic, you need to add `Hask Hask Hask` to the end of all the Bifunctors.)

Function operators. By convention, functions are $f, g, h \ldots$ on the left and `f :: a -> b, g :: a' -> b', h...` on the right (with types unified as appropriate).

```
f † g ⇔ bimap f g :: Bifunctor t => t a a' -> t b b'
f_F ⇔ fmap f :: Functor f => f a -> f b
f ‖ g ⇔ f *** g :: (a, a') -> (b, b')
    where f *** g = \(x, x') -> (f x, g x')
π̀ ⇔ fst :: (a, b) -> a
π́ ⇔ snd :: (a, b) -> b
f △ g ⇔ f &&& g :: a -> (b, b')        -- a = a'
    where f &&& g = \x -> (f x, g x)
```

```
Δx ⇔ double :: a -> (a, a)
    where double x = (x, x)
f | g ⇔ asum f g :: Either a a' -> Either b b'
    where asum f g (Left x)  = Left (f x)
          asum f g (Right y) = Right (g y)
ì ⇔ Left :: a -> Either a b
í ⇔ Right :: b -> Either a b
f ▽ g ⇔ either f g :: Either a a' -> b        -- b = b'
∇x ⇔ extract x :: a
    where extract (Left x) = x
          extract (Right x) = x
f → g ⇔ (f --> g) h = g . h . f
    (-->) :: (a' -> a) -> (b -> b') -> (a -> b) -> a' -> b'
g ← f ⇔ (g <-- f) h = g . h . f
    (<--) :: (b -> b') -> (a' -> a) -> (a -> b) -> a' -> b'
(f ←F g) ⇔ (g <-*- f) h = g . fmap h . f
    (<-*-) :: Functor f => (f b -> b') -> (a' -> f a) -> (a -> b) -> a' -> b'
f_I ⇔ id f :: a -> b
fD ⇔ const id f :: a -> a
x_(FG) ⇔ (fmap . fmap) x
VOID ⇔ const ()
μf ⇔ fix f
```

Now, let's look at the *abides law*:

$$(f \vartriangle g) \triangledown (h \vartriangle j) = (f \triangledown h) \vartriangle (g \triangledown j)$$

Translated into Haskell, this states:

```
either (f &&& g) (h &&& j) = (either f h) &&& (either g j)
```

Which (to me at least) makes more sense: if I want to extract a value from Either, and then run two functions on it and return the tuple of results, I can also split the value into a tuple immediately, and extract from the either "twice" with different functions. (Try running the function manually on a `Left x` and `Right y`.)

- May 26, 2010
- Haskell, Math

7 Responses to “Bananas, Lenses, Envelopes and Barbed Wire A Translation Guide”

1.  *Sjoerd Visscher* says:
   May 26, 2010 at 9:12 am

   I agree, they went a bit overboard with the symbols.

   I think the type of (<-*-) should be Functor f => (f b -> b’) -> (a’ -> f a) -> (a -> b) -> a’ -> b’

2.  *Dougal Stanton* says:
   May 26, 2010 at 10:35 am

   Also the type of `double` should be:

double :: a -> (a, a)

Hoping this renders correctly…

3. *Sean Leather* says:
May 26, 2010 at 10:36 am

See also related work on translating accumulations:
* http://splonderzoek.blogspot.com/2009/09/upwards-and-downwards-accumulations-on.html
* http://github.com/spl/splonderzoek/blob/master/Accumulations.hs

4. *Brent Yorgey* says:
May 26, 2010 at 10:42 am

Awesome, thanks for writing this up! This would have been extremely helpful for me when I read that paper for the first time a few years back… I should probably give it another read this summer.

(LaTeX pro tip: \i and \j produce variants without the dots, which should be used when putting accents over an i or a j.)

5.  *Edward Z. Yang* says:
May 26, 2010 at 11:09 am

Sjoerd, Douglas and Brent, thanks for the corrections, I've updated the post accordingly! (I also took the liberty of editing your comments slightly).

6. *Niklas Broberg* says:
May 26, 2010 at 12:31 pm

A good one, I went through pretty much the same process when reading that paper. Crazy notation.

I believe A_(FG) should translate to "(Functor f, Functor g) => g (f a)"

7.  *Edward Z. Yang* says:
May 26, 2010 at 12:38 pm

Thanks Niklas, it's been fixed. The new translation is a little disingenuous, unfortunately, because the notation in the paper permits A_FGH, whereas in Haskell we have to explicitly parenthesize each.

Leave a Comment

Name (Optional):

Comment:

Post Comment

« Previous Post Next Post »