Applicatives

COMP2209 - Programming III

Dr Julian Rathke
Effects as Structure

• In the previous lecture we concerned ourselves with effectful programming.

• Input/Output was viewed as side-effects to the main computation.

• We represented this at the type level by writing, for any type \( a \), the new type \( \text{IO } a \) that represented an effectful computation that results in a value of type \( a \).

• An interesting conceptual leap to make at this stage is to consider the type \( \text{IO } a \) as an operation on types, similar to lists \( [a] \) or trees \( \text{Tree } a \) or even the option type \( \text{Maybe } a \).

• What these type operations have in common is that they represent values of some base type in some larger world containing extra context or structure around those values.

• We can view Input / Output the same way, it wraps extra context around a value.
Functors - remember them?

• We introduced the concept of Functors way back when we studied trees and generalised mapping
• The class Functor provides a type for

\[ \text{fmap} :: (a \to b) \to f \ a \to f \ b \]

• We map function \( g \) across structure \( f \ a \) to get structure \( f \ b \)
• In fact IO forms a functor:

\[
\text{instance Functor IO where}
\]
\[
- \text{fmap} :: (a \to b) \to \text{IO} \ a \to \text{IO} \ b
\]
\[
\text{fmap} \ g \ \text{mx} = \text{do} \ x \leftarrow \text{mx}
\]
\[
\text{return} \ (g \ x)
\]

Technically we should to prove the functor laws too.
Generalising fmap

- Notice that fmap in Functor class is limited in its type:
  - It accepts only functions of type \((a \rightarrow b)\) and lifts these in to functions that work in the larger context or structure.
- What about a function of type \((a \rightarrow b \rightarrow c)\) say? Perhaps we would like to lift that to be a function of type \(f\ a \rightarrow f\ b \rightarrow f\ c\)
- Or functions of type \((a \rightarrow b \rightarrow c \rightarrow d)\) ? etc
- Clearly we don’t want to add each level of fmap in to the Functor class.
- We can in fact manage with a smarter scheme whereby we leverage currying and just two operations to achieve a range of fmaps
- We need a function called
  
- And an operation
  
\[
\text{pure} :: a \rightarrow f\ a
\]
\[
(\langle*\rangle) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\]
Generalising fmap

- This allows us to define all of our levels of fmap

\[
\begin{align*}
fmap0 & :: a \rightarrow f\ a \\
fmap0 & = \text{pure} \\
fmap1 & :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \\
fmap1\ g\ mx & = \text{pure}\ g\ \langle*\rangle\ mx \\
fmap2 & :: (a \rightarrow b \rightarrow c) \rightarrow f\ a \rightarrow f\ b \rightarrow f\ c \\
fmap2\ g\ mx\ my & = \text{pure}\ g\ \langle*\rangle\ mx\ \langle*\rangle\ my \\
fmap3 & :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow f\ a \rightarrow f\ b \rightarrow f\ c \rightarrow f\ d \\
fmap2\ g\ mx\ my\ mz & = \text{pure}\ g\ \langle*\rangle\ mx\ \langle*\rangle\ my\ \langle*\rangle\ mz \\
\end{align*}
\]

But how are we to understand pure and \langle*\rangle?
Understanding pure :: \( a \rightarrow f \ a \)

- The function pure simply takes a value of some type and embeds it in the larger structure.
- How it does this obviously depends on the structure.

For Lists:
- \( \text{pure } x = [x] \)

For Tree:
- \( \text{pure } x = \text{Leaf } x \)

For Maybe:
- \( \text{pure } x = \text{Just } x \)

For IO:
- \( \text{pure } x = \text{return } x \)

- In each case, there is no real “effect” or extra computation done, the function simply wraps the value in the necessary structure to understand it as the appropriate type.

See I said return is useful.
Understanding \( \langle * \rangle :: f (a \rightarrow b) \rightarrow f a \rightarrow f b \)

- This is a somewhat stranger operation though
- Let’s look at how it works on Lists and see why it makes sense there

\[
gs \langle * \rangle xs = [ g x | g \leftarrow gs, x \leftarrow xs ]
\]

- Given a list of functions (gs::[ a→b ]) and a list of arguments (xs::[a]) for each function we apply it to each argument and return the whole lot as a list.
- Consider the definition of fmap1 on lists as given above in terms of pure and \( \langle * \rangle \),
  - i.e. \( \text{fmap1 } g \ \text{xs} = \text{pure } g \langle * \rangle xs \)
  - The rhs is: \( [g] \langle * \rangle xs = [ g x | g \leftarrow [g], x \leftarrow xs ] \)
  - Which is equivalent to \( [ g x | x \leftarrow xs ] \)
  - Which is what we think of as \( \text{map } g \ \text{xs} \)
Understanding $\langle\star\rangle :: f (a \to b) \to f a \to f b$

Let’s try understand fmap2 on Lists

Given a function $g :: (a \to b \to c)$ and two lists $xs$ $ys$ what would we expect as an answer? Probably $g x y$ for each $x$ in $xs$ and each $y$ in $ys$

Consider the definition of fmap2 on lists as given above in terms of pure and $\langle\star\rangle$,

i.e. $\text{fmap2 } g \quad xs \quad ys = \text{pure } g \langle\star\rangle \quad xs \langle\star\rangle \quad ys$

rhs is: $( [g] \langle\star\rangle \quad xs ) \langle\star\rangle \quad ys$

Which is $( [g x \mid g \gets [g], x \gets xs] ) \langle\star\rangle \quad ys$

Which is equivalent to $[ g x \mid x \gets xs ] \langle\star\rangle \quad ys$

We have a list of (partially applied functions) of type $(b \to c)$

So now, the $\langle\star\rangle$ operation must take each of these partially applied functions and apply each to each value in $ys$

This is exactly what the definition above does.
Understanding <*> on Lists

\[ gs \ <\!*\> \ xs = \ [ \ g \ x \mid g \leftarrow gs, x \leftarrow xs \] \]

- Another way of viewing the List type is as a return value for functions where they may be many possible answers.
- For example, `anagrams :: String \to [\ String ]` allows for many possible anagrams of the input string.
- Suppose we were to define an add operation on Lists of integers `add :: [\ Int ] \to [\ Int ] \to [\ Int ]`
- If we think of Lists as multiple possible return values then the return value here should consider the pairwise addition of each possible value in the input lists.
- And this is exactly what

\[ add \ xs \ ys = \text{pure} (+) <\!*\> xs <\!*\> ys \]
- would achieve.
Understanding <\*\*> - generally

- So the operation <\*\*> :: f (a → b) → f a → f b behaves something like an application operation
- It takes function values that are structured in some context type and arguments to those values, also structured in that context type and applies those functions to the arguments.
- The resulting values are also therefore structured in that same context type.
- We can treat functions as structured functions by embedding them with pure and then using <\*\*> instead of regular application ($)
- So <\*\*> is a *structured* application operator
<*> on Maybe

• Let’s see the definition of <*> for Maybe a

<*> :: Maybe (a → b) → Maybe a → Maybe b

Nothing <*> _ = Nothing
(Just g) <*> mx = fmap g mx

An intuitive reading of this is:
Check if the function g is defined, if not, return Nothing
Check if the argument mx is defined, if not, return Nothing
Otherwise return:  g mx

So what does the following do here?

add :: Maybe Int → Maybe Int → Maybe Int
add xs ys = pure (+) <*> xs <*> ys

Adds two numbers if they are numbers
returns Nothing if either are Nothing!

n.b. This is the same definition as for Lists!
<> on IO

- We saw that pure \( x = \text{return } x \) forms one of the two functions that we use for generalised application at the IO type.
- To define \( mg <> mx \) we simply think
  - There may be IO while evaluating \( mg \) to a pure function
  - There may be IO while evaluation \( mx \) to a pure value
  - We can return the pure value \( mg \) applied to \( mx \)
- Hence

\[
\text{(<>)} :: \text{IO } (a \rightarrow b) \rightarrow \text{IO } a \rightarrow \text{IO } b
\]

\[
mg <> mx = \text{do } g \leftarrow mg
\]

\[
x \leftarrow mx
\]

\[
\text{return } (g \ x)
\]
Example of IO `<*>`

- Let’s implement the function
  - `getChars :: Int → IO String`
- that reads the given number of Chars from stdin
- Hypothetically: to implement this as fetching Chars from a String one might use recursion
- We can follow this idea but use the generalised application to make it type correct

We use `()` as the function to apply embedded in the IO type

```haskell
getChars :: Int → IO String
getChars 0 = return ""
getChars n = pure (:) <*> getChar <*> getChars (n-1)
```

See how `<*>` hides the IO handling!
Applicatives

• The two functions pure and (<*> ) form a useful generalisation principle that we call Applicative structures.

• They are captured as a class in Haskell

```haskell
class Functor f => Applicative f where
    pure :: a -> f a
    (<*> ) :: f (a -> b) -> f a -> f b
```

• Obviously the Functors [a], Maybe a, IO a are all instances of this class.

• As with Functors there are also certain laws that Applicatives are expected to follow.

• Again, these are not compiler checked but you are expected to verify that they hold for any instance of Applicative that you may create.
Applicative Laws

Applicative Law 1:
\[
\text{pure id} \triangleright\!\!<\*\!> \ x = x
\]
The embedding of the identity function is the identity on the structure.

Applicative Law 2:
\[
\text{pure (g x)} = \text{pure g} \triangleright\!\!<\*\!> \text{pure x}
\]
Embedding preserves regular application

Applicative Law 3:
\[
x \triangleright\!\!<\*\!> \text{pure y} = \text{pure (λg → g y)} \triangleright\!\!<\*\!> x
\]
Evaluation order doesn’t matter for pure arguments

Applicative Law 4:
\[
x \triangleright\!\!<\*\!> (y \triangleright\!\!<\*\!> z) = (\text{pure (.)} \triangleright\!\!<\*\!> x \triangleright\!\!<\*\!> y) \triangleright\!\!<\*\!> z
\]
\(<\*\!>\) is associative (ish)
Example : Maybe

- Let’s look at the laws for the Maybe applicative

\[
pure :: a \to Maybe a
\]
\[
pure x = \text{Just } x
\]

\[
\langle*\rangle :: \text{Maybe } (a \to b) \to \text{Maybe } a \to \text{Maybe } b
\]
\[
\text{Nothing } \langle*\rangle _ = \text{Nothing}
\]
\[
(\text{Just } g) \langle*\rangle \text{ mx } = \text{fmap } g \text{ mx}
\]

**Law 1:** \[
pure \text{id } \langle*\rangle x = x
\]
\[
pure \text{id } \langle*\rangle x
\]
\[
= (\text{Just } \text{id}) \langle*\rangle x
\]
\[
= \text{fmap } \text{id } x
\]
\[
= \text{id } x
\]
\[
= x
\]

**Law 2:** \[
pure (g x) = pure g \langle*\rangle pure x
\]
\[
pure (g x)
\]
\[
= \text{Just } (g x)
\]
\[
= \text{fmap } g \text{ (Just } x)
\]
\[
= (\text{Just } g) \langle*\rangle \text{ (Just } x)
\]
\[
= \text{pure } g \langle*\rangle \text{ pure } x
\]
Example: Maybe

• Let’s look at the laws for the Maybe applicative

\[ \text{pure} :: a \rightarrow \text{Maybe} \ a \]
\[ \text{pure} \ x = \text{Just} \ x \]

\[ \text{<>} :: \text{Maybe} \ (a \rightarrow b) \rightarrow \text{Maybe} \ a \rightarrow \text{Maybe} \ b \]
\[ \text{Nothing} \ 	ext{<>} \ _ = \text{Nothing} \]
\[ (\text{Just} \ g) \ 	ext{<>} \ mx = \text{fmap} \ g \ mx \]

Law 3: \[ x \ 	ext{<>} \ \text{pure} \ y = \text{pure} \ (\lambda g \rightarrow g \ y) \ 	ext{<>} \ x \]

\begin{align*}
 x \ 	ext{<>} \ \text{pure} \ y \\
&= (\text{Just} \ x') \ 	ext{<>} \ (\text{Just} \ y) \\
&= \text{fmap} \ x' \ (\text{Just} \ y) \\
&= \text{Just} \ (x' \ y) \\
&= \text{fmap} \ (\lambda g \rightarrow g \ y) \ (\text{Just} \ x') \\
&= \text{fmap} \ (\lambda g \rightarrow g \ y) \ x \\
&= \text{Just} \ (\lambda g \rightarrow g \ y) \ 	ext{<>} \ x \\
&= \text{pure} \ (\lambda g \rightarrow g \ y) \ 	ext{<>} \ x
\end{align*}

Law 3: \[ x \ 	ext{<>} \ \text{pure} \ y = \text{pure} \ (\lambda g \rightarrow g \ y) \ 	ext{<>} \ x \]

\begin{align*}
 x \ 	ext{<>} \ \text{pure} \ y \\
&= \text{Nothing} \ 	ext{<>} \ (\text{Just} \ y) \\
&= \text{Nothing} \\
&= \text{fmap} \ (\lambda g \rightarrow g \ y) \ \text{Nothing} \\
&= \text{Just} \ (\lambda g \rightarrow g \ y) \ 	ext{<>} \ \text{Nothing} \\
&= \text{pure} \ (\lambda g \rightarrow g \ y) \ 	ext{<>} \ \text{Nothing}
\end{align*}
Example: Maybe

Let’s look at the laws for the Maybe applicative

\[
\text{pure} :: a \rightarrow \text{Maybe } a
\]
\[
\text{pure } x = \text{Just } x
\]

\[
\text{<*>} :: \text{Maybe } (a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b
\]
\[
\text{Nothing <*> _ = Nothing}
\]
\[
(\text{Just } g) <*> mx = \text{fmap } g mx
\]

Law 4: \(x <*> (y <*> z) = (\text{pure } (\cdot)) <*>(x <*> y) <*> z\)

\[
x <*>(y <*> z)
= (\text{Just } x’) <*>(y <*> z)
= \text{fmap } x’ (y <*> z)
= \text{Just } (x’ (y <*> z))
= \ldots
\]

very long
Normal form for applicative style

- By using the laws above we can rewrite any well-typed expression that is built using the pure and <*> operator in to what is known as **applicative style**

- That is, every expression can be written in the form

  \[ \text{pure } g \ <*> \ e_1 \ <*> \ e_2 \ <*> \ \ldots \ <*> \ e_n \]

- Where \( g, e_1, e_2, \ldots, e_n \) are expressions that do not use the pure or <*> functions.

- This justifies the name applicative as this just mimics application but for a structured type.

- In the next lecture we’ll look at a further generalisation of applicative like structures where the function being lifted may itself not be pure.
YOUR QUESTIONS

Next Lecture:
Monads