Lambda abstraction

Since functions are first class entities, we should expect to find some notation in the language to create them from scratch. You can view this notation as just filling a gap in the language; mostly we create functions in the usual manner, by writing a named function definition and applying that name to some values. There are times when it is handy to just write a function “inline”. The notation is:

\( \lambda x \ y \rightarrow x + y \)

The notation is somewhat stolen from mathematics; if you are familiar with lambda-calculus then imagine \( \lambda \) instead of the backslash, it might help. Since these values are themselves functions, we just apply them to values to compute something

\[
> (\lambda x \ y \rightarrow x+y) \ 1 \ 2
\]

3

Toolkit

One common way higher order functions are employed is the writing of higher-order abstractions. We have encountered two such abstractions already, embodied in the list comprehension syntax (mapping and filtering):

\[
[ f \ x \mid x \leftarrow xs ] \\
[ x \mid x \leftarrow xs, \ p \ x ]
\]
Both of these operations can be written as general higher-order functions:

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (x:xs) &= (f x):\text{map } f \; xs
\end{align*}
\]

\[
\begin{align*}
\text{filter } p \; [] &= [] \\
\text{filter } p \; (x:xs) &= \begin{cases} 
  x : \text{filter } p \; xs & \text{if } p \; x \\
  \text{filter } p \; xs & \text{otherwise}
\end{cases}
\end{align*}
\]

The combination of the two is natural

\[
[ f \; x \mid x \leftarrow xs, \; p \; x ]
\]

\[
\text{mapfilter } f \; p = (\text{map } f) \circ (\text{filter } p)
\]

The toolkit provides a number of higher order abstractions we can identify. For instance (sticking to lists):

\[
\begin{align*}
\text{sum } [] &= 0 \\
\text{sum } (x:xs) &= x + \text{sum } xs
\end{align*}
\]

\[
\begin{align*}
\text{product } [] &= 1 \\
\text{product } (x:xs) &= x \ast \text{product } xs
\end{align*}
\]

\[
\begin{align*}
\text{and } [] &= \text{True} \\
\text{and } (x:xs) &= x \&\& \text{and } xs
\end{align*}
\]

In each case we can identify: a base case for the empty list, and an operator being applied between the elements.

\[
\begin{align*}
\text{sum } [1,2,3,4] &= 1+2+3+4+0 \\
\text{foldl } (+) \; 0 \; [1,2,3,4,5] &= 1 + (2 + (3 + (4 + (5 + 0)))) \\
\text{foldrl } (+) \; 0 \; [1,2,3,4,5] &= (((0 + 1) + 2) + 3) + 4 + 5 \\
\text{reverse } &= \text{foldl } (\text{\textbackslash reverse} \Rightarrow \text{reverse:rev}) \; []
\end{align*}
\]
Another example of a useful pattern that can be captured by a higher order function is iteration. For a given function we can produce an infinitely long list of values by applying the function to a value to produce the first item, applying the function to that result to get the second item, and so on.

\[ x, f(x), f(f(x)), f(f(f(x))) \ldots \]

This notion is captured by the `iterate` library function:

```haskell
> iterate (2*) 1
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ...]

> take 10 (iterate (2*) 1)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

\[
\text{iterate} :: (a \to a) \to a \to [a]
\]

\[
\text{iterate } f \ x = \ x : \text{iterate } f \ (f \ x)
\]