15–150: Principles of Functional Programming
*Origami (or: how to **fold**)*

Frank Pfenning

Spring 2020

This notes is an attempt to demystify the **fold** family of functions.

Early in the study of functional programming, when advancing to higher-order functions, there are a number of families of functions that are common across varies data types. One of them is **map**, which in my experience is relatively easy to understand, the other is **fold**, which seems more mysterious. But it doesn't have to be! There is a simple geometric intuition when combined with a careful analysis of the types makes it pretty straightforward.

We now show how to visualize and derive various instances of functions in the **fold** family, hopefully demystifying them in the process.

1 Fold on Lists

The **fold** we discuss here is `List.foldr` or just `foldr` in SML. Its sibling function `List.foldl` is specific to lists, while `foldr` is an instance of a generic function that applies to many different types. Personally, I tend to think of `List.foldl` as just `List.foldr` on the reverse of a list.

> In the remainder of this note, when we write `fold` as a function on lists we mean the Standard ML function `foldr`.
>
> ```
> val fold = List.foldr
> ```

Even though it is not the simplest instance, let's start with lists. In SML, we have a predefined type

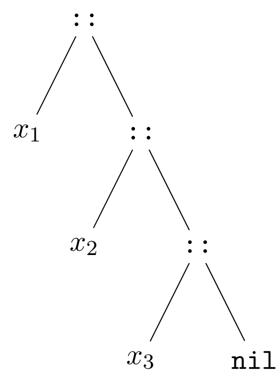
```
datatype 'a list = nil | :: of 'a * 'a list
infixr ::
```

From this, we can easily extract the types of the *constructors*:

```
nil : 'a list
(op ::) : 'a * 'a list -> 'a list
```

As a reminder, `(op` *inf*`)` allows us to use infix operator *inf* as if it were a regular function or constructor. As another reminder, a *constructor* allows us to create expressions (as in `1::(2+3)::nil`) but it can also occur in a *pattern* so we can match against values of the type (as in `case L of nil => "empty" | x::xs => "nonempty"`.

We can visualize a list of type t `list` with elements $x_1 : t$, $x_2 : t$ and $x_3 : t$ with the following diagram.

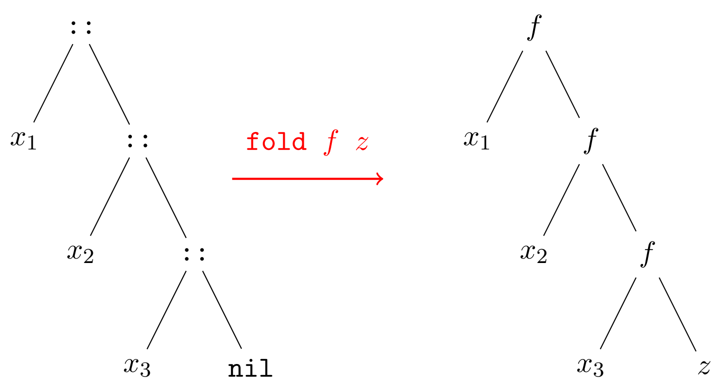


We are now interested in defining functions t `list ->` s for different types t and s. For example:

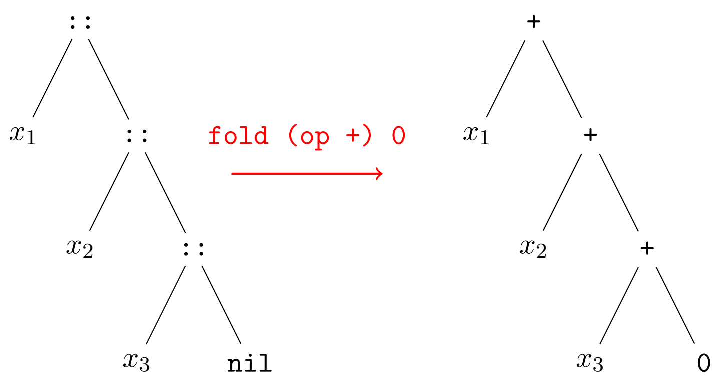
```
val sumList : int list -> int
val maxList : int list -> int
val length : 'a list -> int
val concat : ('a list) list -> 'a list
val map : ('a -> 'b) -> ('a list -> 'b list)
val reverse : 'a list -> 'a list
```

We will see how each of these can be programmed with a use of **fold**.

To see how **fold** works, we visualize replacing every data constructor with a function or constant, depending on the type. In the example of lists, we have one constructor with arguments (`::`) and a constant constructor (`nil`). By choosing different functions/constants for the constructors we can then implement the different functions, such as the ones shown above. The generic diagram is



Before we analyze this picture in more generality, let's consider the case of `sumList`. Clearly, if the function f adds two integers, and the constant z is the integer 0, then `fold` f z will sum up all the elements in the list:



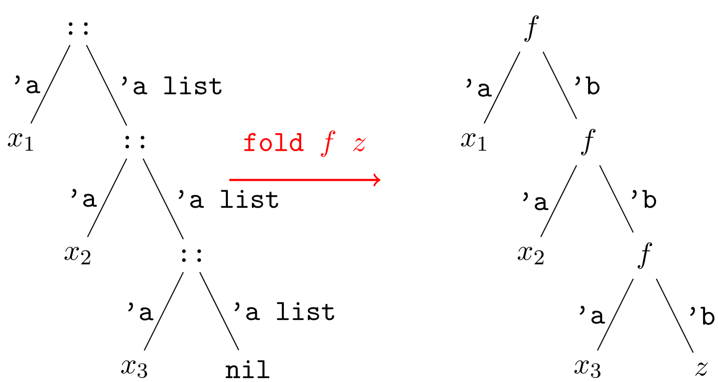

```
(* sumList : int list -> int *)
val sumList = fold (op +) 0
```

In this example, we have f : `int * int -> int` and z : `int`.

Let's now return to the general picture and show the type of each expression above it. We try to make the result of the `fold` as general as possible, that is, we want `fold` f z : `'a list -> 'b`. From the picture we can read off that f : `'a * 'b -> 'b` and z : `'b` if we want `fold` f z : `'a list -> 'b`. Therefore

```
fold : ('a * 'b -> 'b) -> 'b -> ('a list -> 'b)
```

where the second pair of parentheses is optional (but not the first!).



Thinking about it more textually, to derive the types of f and z (which in turn determine the type of `fold`), we replace the type `'a list` in the constructors by `'b`. That's because we want `fold` f z to return a function of type `'a list -> 'b`.

```
(op ::) : 'a * 'a list -> 'a list    (* f : 'a * 'b -> 'b *)
    nil : 'a list                    (* z : 'b *)
```

As a second example, consider computing the maximum of a list of nonnegative integers (defined as -1 if the list is empty). In this example we substitute `int / 'a` and `int / 'b` with `f = Int.max : int * int -> int` and `z = ~1 : int`.

```
(* maxList : int list -> int
 * REQUIRES x >= 0 for all x in L
 * ENSURES maxList L = max{x | x in L or x = ~1} *)
val maxList = fold Int.max ~1
```

Here, it may help to remember the slogan "*functions are values*". Alternatively (and equivalently), we could have defined

```
fun maxList L = fold Int.max ~1 L
```

When the result of applying `fold` f z is polymorphic, we may need to add `L` as an argument as shown here in order to circumvent the so-called *value restriction* in Standard ML.

Computing the length of the list is another interesting example, because the type of the elements doesn't matter (`'a / 'a`) but the result is an integer (`int / 'b`). So we need to find f and z such that

f `: 'a * int -> int`
z `: int`

where f adds 1 to its second argument and z is the length of the empty list (since it is substituted for `nil`).

```
fun length L = fold (fn (x,n) = n+1) 0 L
```

1.1 Concat

Let's see if we can program concatenation

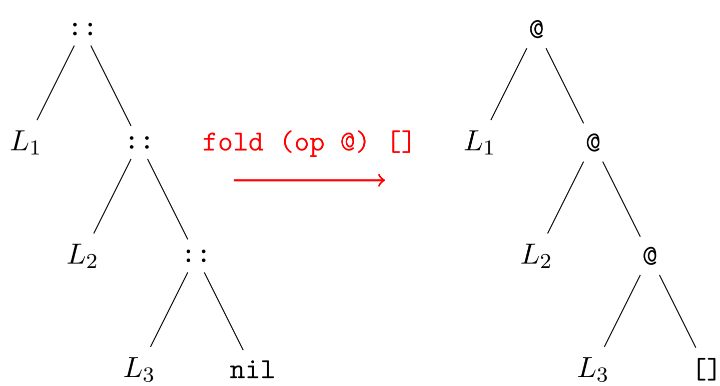
```
(* concat : ('a list) list -> 'a list
 * REQUIRES true
 * ENSURES concat [L1,...,Ln] = L1 @ L2 @ ... @ Ln
 *)
```

To see it as an instance of `fold` we see the type of elements has to be `'a list` while the type of the result has to be `'a list`. So we substitute `'a list / 'a` and `'a list / 'b` and get

(* f `: 'a list * 'a list -> 'a list *)`
(* z `: 'a list *)`
`fold` f z `: ('a list) list -> 'a list`

The constant z stands if for `nil`, so we need z `= []`.[1] Looking at the ensures clause we see that f `= (op @)`, which has the required type. So:

```
fun concat Ls = fold (op @) [] Ls
```



This code is also good (from the efficiency perspective since the append operations only copy what is necessary (L_1, L_2, and L_3).

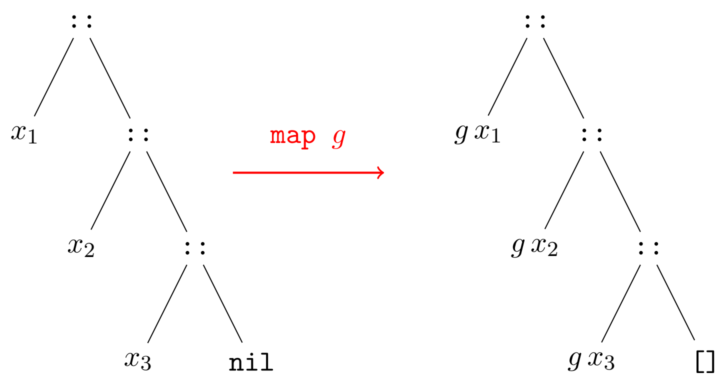
[1] We could equally well write `nil` here, but we reserve it for the constructor we replace as a purely stylistic choice.

1.2 Map

Somewhat trickier is mapping a function g over a list. In general, **map** applies a given function to every element in a data structure but otherwise leaves its structure intact. We have

```
map : ('a -> 'b) -> ('a list -> 'b list)
map g : 'a list -> 'b list
```

Pictorially, the action of map is



Analyzing the type of `map` g for g `: 'a -> 'b` we see that `'a` is arbitrary but we need to substitute `'b list / 'b` in the types of f and z.

f `: 'a * 'b list -> 'b list`
z `: 'b list`

We see that f takes an element x and the already transformed list (of type `'b list`), applies g to x and constructs the new list.

f `: 'a * 'b list -> 'b list = fn (x, ys) => (g x)::ys`
z `: 'b list                 = []`

and therefore

```
fun map g = fold (fn (x,ys) => (g x)::ys) []
```

1.3 The Implementation of fold

We maybe should have done this before, but how do we actually implement `fold`? This is straightforward by pattern matching, again just keeping the picture in mind.

```
(* fold : ('a * 'b -> 'b) -> 'b -> ('a list -> 'b)
 * REQUIRES true
 * ENSURES fold f z [x1,...,xn] = f(x1, f(x2, ... f(xn, z)))
 *)
fun fold f z nil = z
  | fold f z (x::xs) = f(x, fold f z xs)
```

When the implementation of a function h `: 'a list -> 'b` as a fold isn't immediately obvious and the picture doesn't help, we can also try to write it in a specific schematic textual form that can then be translated into a fold. This form is

```
fun h nil = z
  | h (x::xs) = f(x, h xs)
```

where it is important that f does not otherwise refer to `x` or `xs` or h.

First, let's follows this process with `naiveRev` (which is not tail recursive).

```
fun naiveRev nil = []
  | naiveRev (x::xs) = (naiveRev xs) @ [x]
```

Actually, it already has the form we want because the right-hand side in the second clause depends only on `naiveRev xs` and `x`. To make it even more explicit, we can rewrite this to

```
fun naiveRev nil = []
  | naiveRev (x::xs) = (fn (x',ys) => ys @ [x']) (x, naiveRev xs)
```

Here, we have renamed bound variable of `x` to `x'` in the function to avoid any confusion between variable names. Reading off the solution, we obtain

```
f : 'a * 'a list -> 'a list = fn (x',ys) => ys @ [x']
z : 'a list                 = []
```

and

```
fun naiveRev L = fold (fn (x,ys) => ys @ [x]) [] L
```

The more efficient tail-recursive reverse is more difficult to analyze. We have

```
fun revAppend nil acc = acc
  | revAppend (x::xs) acc = revAppend xs (x::acc)
fun reverse L = revappend L []
```

This doesn't quite have the right form, but we can "desugar" the curried patterns into explicit functions

```
fun revAppend nil = fn acc => acc
  | revAppend (x::xs) = fn acc => revAppend xs (x::acc)
fun reverse L = revAppend L []
```

First the types: in the type of fold, `'a` remains unchanged, but we substitute `'a list -> 'a list / 'b` because that's what `revAppend` L returns.

```
f : 'a * ('a list -> 'a list) -> ('a list -> 'a list)
z : 'a list -> 'a list
```

Now we can read off f and z, where z is easy

```
z : 'a list -> 'a list = (fn acc => acc)
```

Remembering that application is left-associative, the right-hand side in the second clause is the same as `fn acc => (revAppend xs) (x::acc)` so, indeed, the right-hand side depends only on `x` and `revAppend xs`.

```
f : a' * ('a list -> 'a list) -> ('a list -> 'a list)
f = fn (x, accFun) => fn acc => accFun (x::acc)
```

Putting everything together, we have

```
fun revAppend L = fold (fn (x, accFun) => fn acc => accFun (x::acc))
                       (fn acc => acc) L
fun reverse L = revAppend L []
```

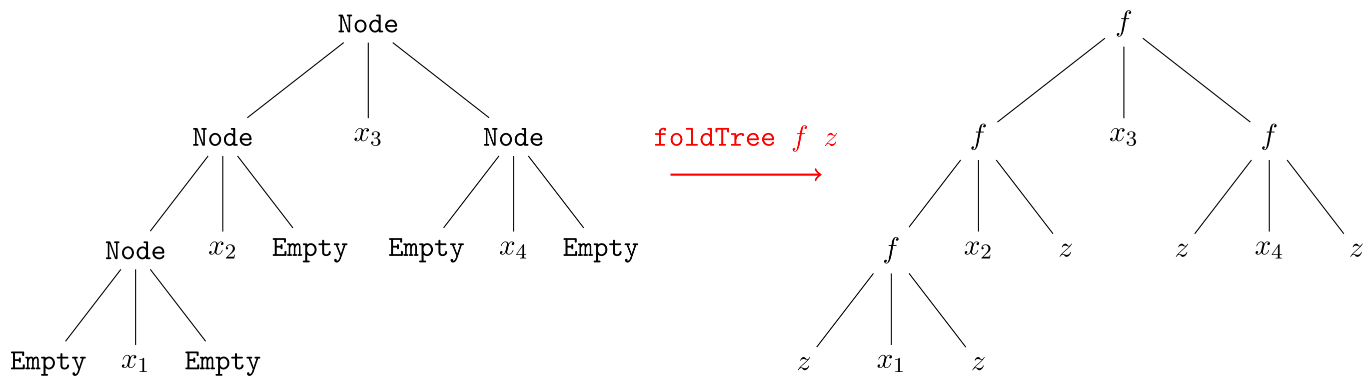
2 Fold on Other Datatypes

Fold is a generic operation that can be defined on just about any datatype.[2] The general schema is to replace the constructors by functions or constants, depending on their type. We first consider trees.

2.1 Binary Trees

```
datatype 'a tree = Node of 'a tree * 'a * 'a tree | Empty
```

Pictorially:



We won't bother drawing in the types, but from

```
Node : 'a tree * 'a * 'a tree -> 'a tree
Empty : 'a tree
foldTree f z : 'a tree -> 'b
```

we deduce

```
f : 'b * 'a * 'b -> 'b
z : 'b
```

For example:

```
val sumTree = foldTree (fn (suml, x, sumr) => suml + x + sumr) 0
```

Let's define a function to create the "mirror image" a tree, by which we mean a reflection of the tree about a vertical axis through the root. Creating the mirror image of a tree means that in the type of f and z we substitute `'a / 'a` and `'a tree / 'b`.

```
f : 'a tree * 'a * 'a tree -> 'a tree
z : 'a tree
```

Referring back to the picture, we see that f just has to swap the left and right subtrees, keeping the element in place, and z is just the empty tree.

```
fun mirror T = fold (fn (ml, x, mr) => Node(mr, x, ml)) Empty T
```

The `foldTree` function is easy to implement with recursion and pattern matching, following the picture.

```
fun foldTree f z (Empty) = z
  | foldTree f z (Node(l,x,r)) = f(foldTree f z l, x, foldTree f z r)
```

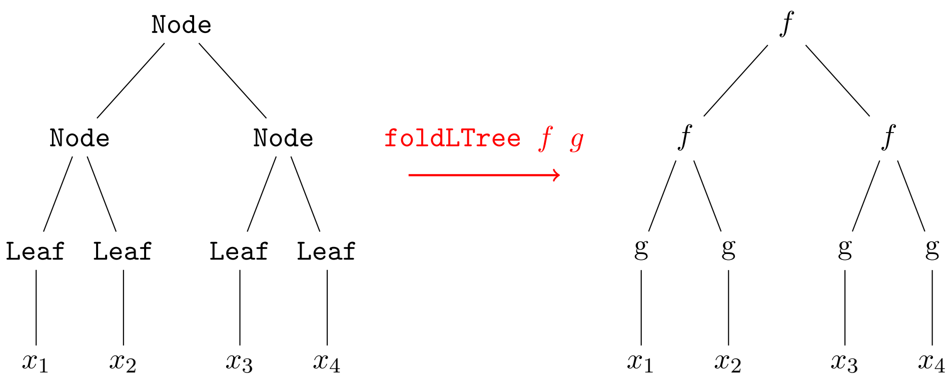
[2] I hesitate only when a `datatype` declaration contains functions.

2.2 Leafy Trees

A variant we have considered is a tree where the data are stored in the leaves.

```
datatype 'a lTree = Node of 'a lTree * 'a lTree | Leaf of 'a
```

Pictorially:



Notice that here we have two constructors (`Node` and `Leaf`) and now constants. Since we want

```
foldLTree : 'a lTree -> 'b
```

we conclude that

`Node : 'a lTree * 'a lTree -> 'a lTree`	f `: 'b * 'b -> 'b`
`Leaf : 'a -> 'a lTree`	g `: 'a -> 'b`

For example, to sum the elements of the leafy tree with integers, the function f just has add up the results from the subtrees and the function g just has to return the value of the integer stored in the leaf.

```
(* sumLTree : int lTree -> int *)
val sumLTree = foldLTree (op +) (fn x => x)
```

Again, the function itself is also easy to implement

```
fun foldLTree f g (Node(l,r)) = f (foldLTree f g l, foldLTree f g r)
  | foldLTree f g (Leaf(x)) = g x
```

2.3 Options

Even though it is not recursive, we can still define fold for a type such as `'a option`

```
datatype 'a option = SOME of 'a | NONE
```

We just replace the constructors with corresponding functions (f) or constants (z).

```
SOME : 'a -> 'a option      f : 'a -> 'b
NONE : 'a option            z : 'b
foldOpt f z : 'a option -> 'b
foldOpt : ('a -> 'b) -> 'b -> 'a option -> 'b
fun foldOpt f z (SOME(x)) = f x
  | foldOpt f z (NONE) = z
```

We see that `foldOpt` applies f to the value v in `SOME(`v`)` and returns a “default” value z in case of `NONE`.

2.4 Natural Numbers

We can also explore **fold** operations on data types that are not polymorphic. Consider, for example, natural numbers in unary form.

```
datatype nat = Succ of nat | Zero
```

The number 3, for example, is represented as

```
Succ(Succ(Succ(Zero))) : nat
```

or in tree form as

$$\begin{tikzcd}
\texttt{Succ} \arrow[d, dash] & & & f \arrow[d, dash] \\
\texttt{Succ} \arrow[d, dash] & {} \arrow[r, "\texttt{foldNat}\ f\ z"] & {} & f \arrow[d, dash] \\
\texttt{Succ} \arrow[d, dash] & & & f \arrow[d, dash] \\
\texttt{Zero} & & & z
\end{tikzcd}$$

In this case, there is no element type `'a` so we just expect

`foldNat` f z `: nat -> 'b`

from which we conclude

f `: 'b -> 'b`
z `: 'b`

and we obtain the definition

```
foldNat : ('b -> 'b) -> 'b -> (nat -> 'b)
fun foldNat f z (Succ(x)) = f (foldNat f z x)
  | foldNat f z (Zero) = z
```

This means that `foldNat` f z n just computes $f(f(\ldots f(z)))$ where f is iterated n times.