15–150: Principles of Functional Programming

Functions are Values

Michael Erdmann*

Spring 2020

1 Outline

- Functions as values
  - anonymous, non-recursive, function expressions
  - recursive function declarations
  - application and composition
  - extensional equivalence
- Functions as values
  - functions that take functions as arguments
  - functions that produce functions as results
- Maps and folds on lists and trees
  - higher-order functions in action
  - bulk processing of data
  - potential for parallelism

*Adapted from a document by Stephen Brookes.

2 Introduction

So far we have largely used functions from tuples of values to values, and values have been integers/lists/trees. We call such functions "first-order". But we can also define functions that take functions as arguments. Those are called "higher-order" functions. And we can also define functions that return functions as results. We have already seen a few such examples, but we are about to see many more.

Recall how we defined *equivalence* for first-order functions. This notion generalizes in a straightforward way to higher-order functions.

SML has a special syntax for "curried functions" of several arguments, which allows us to define functions that return functions using a convenient notation.

These ideas are very powerful. We can write elegant functional programs to solve interesting problems in very natural ways, and we can reason about functional programs by thinking in terms of mathematical functions.

3 Functions as values

Recall that SML allows recursive function definitions, using the keyword `fun`. Function definitions (or declarations) give names to function values. If we want to apply the function to a variety of different arguments, it is convenient to use the name over and over, rather than having to write out the function code every time.

Nonetheless, we can also build "anonymous", non-recursive, function expressions, using `fn`. This can also be convenient, especially when we want to supply the function as an argument to another function! For example, the expression

```
(fn x:int => x+1)
```

is well-typed, has type `int -> int`. This expression (since it begins with `fn`) is called an *abstraction*.[1]

We can use it to build other expressions, by applying it to an expression of the correct argument type, as in

```
(fn x:int => x+1) (20+21)
```

[1] Also known as a λ-expression for historical reasons: `fn x => e` is the SML analogue to $\lambda x.e$ from the λ-calculus. λ is pronounced "lambda".

and this particular expression evaluates to 42.

We can also build other expressions out of abstractions. For example,

```
(fn x:int => x+1, fn y:real => 2.0 * y)
```

is an expression of type `(int -> int) * (real -> real)`.

If we so desire, we can even give a name to a non-recursive function, using a `val` declaration, as in the declaration

```
val (inc, dbl) = (fn x:int => x+1, fn y:real => 2.0 * y)
```

which binds `inc` to the function value `fn x:int => x+1` and `dbl` to the function value `fn y:real => 2.0*y`.

The SML interpreter treats functions as values: evaluation of an expression of type `t->t'` stops as soon as it reaches an abstraction, i.e., a `fn`-expression or a function name that has been declared (and hence bound to) a function value.

Programmers can use functions as values, either as arguments to functions, or as results to be returned by functions.

Composition

Function composition is an infix operator written as `o`. Using the `op` keyword we can also use `o` as a function

```
(op o) : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b).
```

For all types `t1, t2, t3`, when `f:t2 -> t3` and `g:t1 -> t2`, `(f o g)` has type `t1 -> t3`, and satisfies the equivalence

$$\texttt{(f o g)(x)} \cong \texttt{f(g x)}$$

for all values `x` of type `t1`. Some examples of function values built using composition:

```
val inc = fn (x:int) => x+1;
val double = fn (x:int) => 2*x;
val add2 = inc o inc;
val add4 = add2 o add2;
val shrink =  fn (x:int) => case Int.compare(x, 0) of
                              EQUAL => 0
                            | LESS  => x+1
                            |  _    => x-1;
```

Equivalence for basic values and first-order functions

Suppose, for the moment, that we only consider basic types, such as `int`, `bool`, `string`, etc., along with product types and user-defined datatypes built from these basic types. Now let us consider functions on those types, such as functions of type `int -> int` in the code above. We call these functions *first-order* functions because their arguments and results are simple data values, in this case integers, not functions.

For the code given above, the following equivalences hold:

```
inc 2 ≅ 3
(double o inc) 4 ≅ 10
(inc o double) 4 ≅ 9
add2 2 ≅ 4
add4 2 ≅ 6
shrink 2 ≅ 1
shrink (~2) ≅ ~1
(fn y:int => add4(add4 y)) (add2 15 + add4 13) ≅ 42
```

Each of the expressions above has type `int`. Recall, in our discussion of *extensional equivalence* (see Lecture 1), we said that two expressions of type `t` are equivalent if and only if they evaluate to the same value (necessarily of type `t`), or they both raise the same exception, or they both loop forever. That is how one should interpret the equivalences above: each equivalence says that the expression on the left evaluates to the value on the right.

This notion of equivalence extends to first-order functions. We say that two function values `f` and `g` of type `t -> t'` are *extensionally equivalent* if and only if, for all values `v` of type `t`, `f(v)` and `g(v)` are extensionally equivalent. Symbolically, `f ≅ g` iff `f(v) ≅ g(v)` for all values `v:t`.

Here are some examples of equivalence for function expressions of type `int -> int`:

```
(inc o double) ≅ (fn x:int => (2*x)+1)
(double o inc) ≅ (fn x:int => 2*(x+1))
add2 ≅ (fn y:int => inc(inc y))
add4 ≅ (fn z:int => add2(add2 z))
(fn x:int => x+1) ≅ (fn w:int => w+1)
```

4 Higher-order functions

In the previous section we presented some first-order functions from integers to integers. In contrast, the composition function, also discussed above as

```
(op o) : ('a -> 'b) * ('c -> 'a) -> ('c -> 'b)
```

has a (most general) type of the form `t -> t'`, with argument type `t` being

```
('a -> 'b) * ('c -> 'a),
```

built from function types. A function type like this in which the "argument type" is itself a function type, is classified as a *higher-order* type.[2] Observe that every *instance* of a higher-order type is also a higher-order type, since the arrows don't go away.

Consider a higher-order function `f` of type `t -> t'`, with `t` itself being some function type. Applying `f` to a function `g` of the required argument type `t` will produce an application expression `f(g)` with the corresponding result type `t'`, just as with first-order functions. There is nothing special here, as far as SML is concerned. The underlying reason is that an expression with a function type is much like any other expression with a well-defined type, and function values are much like other values: they may be passed as arguments to function calls and/or returned as results from function calls.

Equivalence for higher-order functions

The notion of *extensional equivalence* applies to higher-order function types. Intuitively, this is much as before: two functions are equivalent iff they map equivalent arguments to equivalent results.

There isn't really anything else we need to say, but perhaps we should be careful about base cases when defining equivalence in order to avoid circularity in the definition. The base cases are given by defining equivalence for SML's basic types, just as we did earlier. One can then proceed inductively:

Assume that we already have a definition of equivalence on some collection of types. We may extend that definition to include product types and user-defined datatypes built from that collection of types.

[2] More formally, the order of a type is defined recursively, by regarding basic types to have order 0, then defining the order of a function type by $\mathrm{ord}(\mathtt{t_1 -> t_2}) = \max\{\mathrm{ord}(\mathtt{t_1}) + 1, \mathrm{ord}(\mathtt{t_2})\}$. In this document, "higher-order" means a function with a type whose order is at least 2. Functions that return higher-order functions are therefore also classified as higher-order. Be aware that some authors also consider functions to be higher-order if the return type $\mathtt{t_2}$ is any function type, i.e., has order at least 1.

Now suppose we have a definition of equivalence in particular for expressions of type `t` and expressions of type `t'`. Consider two functions `F` and `G` of type `t -> t'`. We say that `F` $\cong$ `G` if and only if, for all values `x` of type `t`, `F(x)` $\cong$ `G(x)`. Observe as well: Because of referential transparency, this is also the same as saying that `F` $\cong$ `G` iff, for all values `x` and `y` of type `t`, if `x` $\cong$ `y` then `F(x)` $\cong$ `G(y)`. (Recall: *referential transparency* is the property that replacing a sub-expression by an equivalent sub-expression produces an equivalent expression.)

It may not be obvious, but equivalence satisfies the usual mathematical properties one expects for a sensible notion of "equality". In particular, for every type `t`, equivalence for expressions of type `t` is an *equivalence relation*:

- For all expressions `e:t`, `e` $\cong$ `e`.
- For all expressions $\mathtt{e}_1, \mathtt{e}_2 : \mathtt{t}$, if $\mathtt{e}_1 \cong \mathtt{e}_2$, then $\mathtt{e}_2 \cong \mathtt{e}_1$.
- For all expressions $\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3 : \mathtt{t}$, if $\mathtt{e}_1 \cong \mathtt{e}_2$ and $\mathtt{e}_2 \cong \mathtt{e}_3$, then $\mathtt{e}_1 \cong \mathtt{e}_3$.

Referential Transparency

Now that we've spoken more generally about equivalence, we can state a more precisely formulated version of referential transparency, as follows. Here we use the notation $E[e]$ for an expression E having a sub-expression e; then $E[e']$ for the expression obtained by replacing the sub-expression e by e'; and we implicitly refer to equivalence on type `t` (in $e \cong e'$) and on type `T` (in $E[e] \cong E[e']$).

Further, as we have done in class already, here we represent binding of value `v` to identifier `x` by writing `[v/x]`.

> (Referential Transparency)
> If $E[e]$ is a well-typed expression of type `T`, with a sub-expression e of type `t`, and e' is another expression of type `t` such that $e \cong e'$, then $E[e] \cong E[e']$.

Another way to say the same thing is that *equivalence is a congruence*. That's a fancy way to say that "Every syntactic construct in SML *respects* equivalence".

For example, to say that

```
(op +) : int * int -> int
```

respects equivalence means that for all well-typed expressions $\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}'_1, \mathtt{e}'_2$ of type `int`, if $\mathtt{e}_1 \cong \mathtt{e}'_1$ and $\mathtt{e}_2 \cong \mathtt{e}'_2$ then $\mathtt{e}_1 + \mathtt{e}_2 \cong \mathtt{e}'_1 + \mathtt{e}'_2$.

The next page has some simple examples, to help clarify these ideas.

- Consider the function `checkzero:(int -> int) -> int` given by:

  ```
  fun checkzero (f:int -> int):int = f(0)
  ```

  If `f,g:int->int` and `f` $\cong$ `g`, observe that `checkzero(f)` $\cong$ `checkzero(g)`. Hence, `checkzero` $\cong$ `checkzero` according to the definition of equivalence for type `(int -> int) -> int`.

- Recall the composition operator, as discussed above. One may check that, for all types `t1, t2, t3`, if `f,f':t2 -> t3` and `g,g':t1 -> t2`, and `f` $\cong$ `f'` and `g` $\cong$ `g'`, then `f o g` $\cong$ `f' o g'`. This is the same as saying that `(op o)` respects equivalence.

- We saw that expressions `fn x:int => x+x` and `fn y:int => 2*y` are equivalent. It follows that

  ```
  40 + (fn x:int => x+x) 1 ≅ 40 + (fn y:int => 2*y) 1
  ```

  Although in this particular example we can see that the value of the overall expression is 42 in both cases, in a more complex expression we could make the analogous deduction even without knowing what the value of the entire expression is!

Before moving on, notice that *equivalence* on function expressions is *NOT* the same as "both evaluate to the same value, or both raise the same exception, or both loop forever, which was how we characterized equivalence on basic types like `int` and `int list`. Instead, two function expressions `e` and `e'` are equivalent iff they both loop forever or they both raise the same exception or they evaluate to function values `f` and `f'` that are extensionally equivalent.

For example, the expressions `fn x:int => 2*x` and `fn y:int => y+y` are (already) function values (of type `int -> int`) that are extensionally equivalent. But they don't evaluate to the *same* function value: each evaluates to itself and the expressions are syntactically different. Nevertheless, each of these expressions can be said to *compute* the same function from integers to integers, and that's why we are comfortable saying that they are equivalent.

Equivalence and evaluation to a value

For each type `t` there is a set of SML values (or "syntactic values") of type `t`. We can connect SML evaluation-to-a-value (written `e` $\hookrightarrow$ `v`, where `v` is an SML value) and extensional equivalence (written `e` $\cong$ `e'`), as follows:

- If `e:t` and `e` $\hookrightarrow$ `v`, then `e` $\cong$ `v` (according to our notion of equivalence on type `t`).
- If `e` $\cong$ `e'` (both of type `t`) and there is an SML value `v` (of type `t`) such that `e` $\cong$ `v`, then there is an SML value `v'` (again, of type `t`) such that `e'` $\hookrightarrow$ `v'` and `v` $\cong$ `v'`.

Make sure you notice that (and understand why) we did *not* say here that if `e` $\cong$ `e'` and `e` $\cong$ `v` then `e'` $\hookrightarrow$ `v`. (Such a conclusion would be correct for so-called *observable values*, such as values of type `int`, `string`, `bool`, etc., but it is not true for all types of values.)

Recursive function definitions and equivalence

In the scope of a recursive function definition of the form

```
fun f(x:t):t' = e'
```

the function name `f` satisfies the equivalence `f` $\cong$ `fn x:t => e'`.

We also have this equivalence law:

If `v` is a value such that `e` $\cong$ `v`, then

`(fn x:t => e') e` $\cong$ `[v/x]e'`.

We have thus far interpreted `[v/x]` as an environment binding. However, from the perspective of extensional equivalence, it is also legitimate to interpret `[v/x]e'` as the expression built by substituting `v` for the free occurrences of `x` in `e'`. In that form, this law is sometimes called the *β-value reduction law*, for historical reasons having to do with the λ-calculus. Note: Since SML functions are call-by-value, i.e., functions always evaluate their arguments, one cannot substitute into a function body unless the argument expression has been evaluated down to a syntactic value.

Functions evaluate their arguments

Let `(fn x:t => e')` be well-typed expression of type `t->t'`. If `e:t` is a well-typed expression and `v` is an SML value such that `e` $\hookrightarrow$ `v`, then

`(fn x:t => e') e ==> [v/x]e'`.

Looping forever and equivalence

We have been very careful in the above account of evaluation and equivalence to avoid accidentally concluding erroneous "facts" by sloppy reasoning. For example, consider the recursive function given by

```
fun silly (n:int) : int = silly n
```

This declaration binds `silly` to the function value `(fn n:int => (silly n))`, of type `int -> int`. As we said above, in the scope of this declaration, we have the equivalence

(*) `silly` ≅ `(fn n:int => (silly n))`

What can we prove about the "value" of the expression `silly 0`? (It isn't hard to see that evaluation of this expression loops forever.) We can make a sequence of logical steps, such as:

```
silly 0
   ≅ (fn n:int => (silly n)) 0    by (*)
   ≅ [0/n] (silly n)              by the value-reduction law
   ≅ silly 0
```

but one *cannot* find a value `v` such that `silly 0` ≅ `v` is provable!

5 Higher-order functions on lists

Filtering a list

A total function `p` of type `t -> bool` represents a "predicate" on values of type `t`. A value `v:t` such that `p(v) ==> true` is said to *satisfy* the predicate; when `p(v) ==> false` the value `v` does not satisfy the predicate. Totality of `p` means that for all values `v` there is a definite answer.

We can define a recursive SML function with the following specification:

```
(* filter : ('a -> bool) -> ('a list -> 'a list)                    *)
(* REQUIRES: p is a total function                                  *)
(* ENSURES: filter p L ==> a list of the values in L that satisfy p, *)
(*                         in the same order as in L.               *)

fun filter p [ ] = [ ]
  | filter p (x::L) = if p(x) then x::filter p L else filter p L
```

Actually some of the parentheses that we included in the type specification above for `filter` are superfluous, because the SML function type constructor `->` associates to the right. We could just as well have said

```
(* filter : ('a -> bool) -> 'a list -> 'a list                      *)
```

The only reason we didn't do that was because we wanted to emphasize the fact that, when applied to a predicate `p`, `filter p` returns a *function*. You don't have to always apply `filter` to a predicate and a list. It's perfectly reasonable to apply it to just a predicate and to use the resulting function as a piece of data to be passed around, used by other functions, or just returned.

Consider for example the following first-order function

```
(* divides : int * int -> bool *)
(* REQUIRES: x>0 *)
(* ENSURES:
    For all y>0, divides(x, y) ==> true if y mod x = 0, false otherwise *)

fun divides (x, y) = (y mod x = 0)
```

Because of its type, the only way we can apply this function is to a pair of integers. For example, `divides(2,4)` $\hookrightarrow$ `true`.

But if we introduce a closely related function as follows, with the type `int -> (int -> bool)` we will be able to "partially apply" it to one integer and get back a predicate that checks for divisibility by that integer.[3]

```
(* divides' : int -> (int -> bool) *)
(* REQUIRES: x>0 *)
(* ENSURES:
    For all y>0, divides' x y ==> true if y mod x = 0, false otherwise *)

fun divides' x y = (y mod x = 0)
```

For example, we can apply `divides'` to 2 and get a function that tests for evenness:

```
(* even : int -> bool *)
val even = divides' 2
```

We can use `divides'` to help us build lists of prime numbers:

```
(* sieve : int list -> int list *)
fun sieve [ ] = [ ]
  | sieve(x::L) = x::sieve(filter (not o (divides' x)) L)

(* primes : int -> int list *)
fun primes n = sieve (upto(2, n))
```

Here we have used the built-in SML function

```
not : bool -> bool
```

whose behavior is completely determined by

```
not true ==> false
not false ==> true
```

In addition, `upto : int * int -> int list` is the function given by

```
fun upto(i:int, j:int):int list = if i>j then [ ] else i::upto(i+1,j)
```

[3]This is an example of "currying" a function of two arguments to obtain a function of one argument that returns another function. We will return to this idea at a later date when we talk about "staging" computation.

Thus `upto(2, n)` returns the list of integers from `2` to `n`, inclusive. So, for `n>1`, `primes n` returns the list of prime numbers between 2 and `n`, in increasing order. For example, `primes 10 = [2,3,5,7]`.

Now, admittedly, we *could* have developed the primes function and the sieve function differently, without insisting on the use of `divides'` and `filter`. But we feel this is an elegant example that shows the potential advantages of functional programming with higher-order functions.

Transforming the data in a list

A very common task involves taking a list of some type and performing some operation on each item in the list, to produce another list, of a possibly different type. Typically, we have some function that we wish to apply to all items in the list. We can encapsulate this process very naturally as a higher-order function

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

Again, some of the parentheses are redundant here, but we want to emphasize that `map` takes a function as argument and returns a function as its result. SML has a built-in function like this, and its definition is:

```
fun map f [ ] = [ ]
  | map f (x::L) = (f x)::(map f L)
```

So `map f [] ≅ []`
and `map f [`x_1`, ..., `x_n`] ≅ [f(`x_1`), ..., f(`x_n`)]`.

As a simple example using `map`, the function

```
addtoeach' : int -> (int list -> int list)
```

defined by

```
fun addtoeach' a = map (fn x => x+a)
```

is the "curried" version of the function

```
addtoeach : int * int list -> int list
```

defined by

```
fun addtoeach (x:int, L:int list) =
   case L of
         [ ]  => [ ]
     | (y::R) => (x+y)::addtoeach(x,R)
```

Consider now the function `sum:int list -> int` given by:

```
fun sum [ ] = 0
  | sum (x::L) = x + sum L
```

For an integer list `L`, `sum L` returns the sum of the integers in `L`. We can use `sum` and `map` to obtain a function

```
count : int list list -> int
```

that adds all the integers in a list of integer lists:

```
fun count R = sum (map sum R)
```

In reasoning about code that uses `map`, we can typically use induction on the length of lists.

For example, as an exercise, prove the following properties:

- If `f:t1 -> t2` is total, then for all `L:t1 list`, `map f L` evaluates to a list `R` such that `length(L) = length(R)`.
- If `f:t1 -> t2` is total, then for all lists `A` and `B` of type `t1 list`,

  `map f (A @ B)` $\cong$ `(map f A) @ (map f B).`

  The assumption that the function being "mapped along the lists" is total was made here to simplify the problem. (We could also simplify the problem by merely assuming that `f(x)` reduces to a value for all values `x` that occur in the lists.) The result holds more generally.

Combining the data in a list

Another common task involves *combining* the values in a list to obtain some composite value: for example, as we have seen, adding the integers in a list to obtain their sum. Other examples include concatenating a list of lists to get a single list, and counting the number of items in a list (to get its length). Again we can encapsulate the general idea using higher-order functions. Since lists are inherently sequential data structures there are actually two distinct and natural sequential orders in which we might combine items: from head to tail, or *vice versa*. SML has two built-in functions, accordingly:

```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

and their definitions are

```
fun foldl g z [ ]    = z
  | foldl g z (x::L) = foldl g (g(x,z)) L

fun foldr g z [ ]    = z
  | foldr g z (x::L) = g(x, foldr g z L)
```

For all types `t1` and `t2`, all functions `g:t1 * t2 -> t2`, and all values `z:t2`,

```
foldl g z : t1 list -> t2
foldr g z : t1 list -> t2
```

are functions that can be applied to a list and will combine (using `g`) the items in the argument list with `z`. Assuming that `g` is total, for all integers $n > 0$ and all values $\mathtt{x}_1, \ldots, \mathtt{x}_n$ of type `t1`, the following equivalences hold:

$$\begin{array}{lcl} \texttt{foldl g z [x}_1\texttt{,...,x}_n\texttt{]} & \cong & \mathtt{g}(\mathtt{x}_n, \mathtt{g}(\mathtt{x}_{n-1}, \ldots, \mathtt{g}(\mathtt{x}_1, \mathtt{z}) \ldots)) \\ \texttt{foldr g z [x}_1\texttt{,...,x}_n\texttt{]} & \cong & \mathtt{g}(\mathtt{x}_1, \mathtt{g}(\mathtt{x}_2, \ldots, \mathtt{g}(\mathtt{x}_n, \mathtt{z}) \ldots)). \end{array}$$

For the empty list, `foldl g z [] ≅ z` and `foldr g z [] ≅ z`.

As examples, we can add the integers in an integer list using either of these fold functions:

```
fun suml L = foldl (op +) 0 L
fun sumr L = foldr (op +) 0 L
```

And we'll get, for all $n \geq 0$ and all integer lists $[\mathtt{x}_1, \ldots, \mathtt{x}_n]$,

$$\begin{array}{lcl} \texttt{suml [x}_1\texttt{,...,x}_n\texttt{]} & \cong & \mathtt{x}_n + (\mathtt{x}_{n-1} + \ldots + (\mathtt{x}_1 + 0)\ldots) \\ \texttt{sumr [x}_1\texttt{,...,x}_n\texttt{]} & \cong & \mathtt{x}_1 + (\mathtt{x}_2 + \ldots + (\mathtt{x}_n + 0)\ldots) \end{array}$$

Since addition of integers is an associative and commutative operation, the sums on the right-hand sides of these equivalences are equal. Since `x+0` $\cong$ `x` for all integers `x`, we therefore have

$$\texttt{suml [x}_1\texttt{,...,x}_n\texttt{]} \cong \textstyle\sum_{i=1}^{n} \mathtt{x}_i \cong \texttt{sumr [x}_1\texttt{,...,x}_n\texttt{]}.$$

And since every value of type `int list` is expressible in the form $\texttt{[x}_1\texttt{,...,x}_n\texttt{]}$ for some $n \geq 0$ and some integer values $\mathtt{x}_i$ $(i = 1, \ldots, n)$, this shows that the functions `suml` and `sumr` are extensionally equivalent, i.e., `suml` $\cong$ `sumr`.

Given our earlier comments about `sum` (as defined on page 13), we have also shown here that `suml` $\cong$ `sum` $\cong$ `sumr`.

It follows, then, that the functions

```
fun countl R = suml (map suml R)
fun countr R = sumr (map sumr R)
```

are also extensionally equivalent. Moreover, `countl` $\cong$ `countr` $\cong$ `count`.

Caution: Functions `foldl` and `foldr` are *not* extensionally equivalent. To see this, compare `foldl (op -) 0 L` with `foldr (op -) 0 L`.

Exercise

Suppose `g` is a total function of type `t1 * t2 -> t2`, such that for all values `x:t1, y:t1, z:t2`, `g(x, g(y, z))` $\cong$ `g(y, g(x, z))`. Prove that for all values `z:t2` and lists `L:t1 list`,

$$\texttt{foldr g z L} \cong \texttt{foldl g z L}.$$

Explain how this result, along with the given assumptions, implies that `foldr g` $\cong$ `foldl g`.

Insertion sort, revisited

Recall that we defined a function

```
ins : int * int list -> int list
```

with the specification that for all integer values `x` and all integer list values `L`, if `L` is sorted, then `ins(x,L)` reduces to a sorted permutation of `x::L`.

Using this function, we can obtain an implementation of insertion sort, by defining:

```
(* isortl : int list -> int list *)
val isortl = foldl ins [ ]
```

Indeed, one may prove, by induction on `L`, that for all integer list values `L`,

`foldl ins [ ] L` $\cong$ `a sorted permutation of L.`

(In this proof the only information about `ins` that you need is the prior result that `ins` satisfies its specification.)

Equally well we could have used the other fold function:

```
(* isortr : int list -> int list *)
val isortr = foldr ins [ ]
```

And we could prove, by induction on `L`, that for all integer list values `L`,

`foldr ins [ ] L` $\cong$ `a sorted permutation of L.`

Further, since for an integer list `L` there is *exactly one* sorted permutation of `L` (assuming we don't distinguish between identical duplicate integers), we could then conclude that `isortl` $\cong$ `sortr`.

Exercise

If you did the exercise above, you can use its result here.
Using the definition of `ins`, show that for all values `x,y:int` and sorted `L:int list`, `ins(x, ins(y, L))` $\cong$ `ins(y, ins(x, L))`.
Deduce that `isortl` $\cong$ `isortr`.

6 Higher-order functions on trees

Lists are just one way to collect data into a data structure. All of the ideas in the previous sections can be adapted to deal with other data structures, such as trees. The ideas of performing an operation on all items in a list, and combining all the items in a list, generalize to many other settings: any time we have a collection of data. Here we will discuss some analogous operations invoking the (parameterized) datatype of binary trees. We will see later that

every time we introduce a new datatype for structured collections of data there will be a way to design analogous operations.

Recall the parameterized datatype definition for binary trees with values at the internal nodes.

```
(* Binary trees with values at internal nodes *)
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* Empty : 'a tree *)
(* Node : 'a tree * 'a * 'a tree -> 'a tree *)
```

Filtering a tree

Given a predicate `p:t -> bool` and a tree value `T: t tree`, we can build a tree consisting of the items in `T` that satisfy `p`.

Give specs for the following functions:

```
(* insert : 'a * 'a tree -> 'a tree *)
fun insert (x, Empty) = Node(Empty, x, Empty)
  | insert (x, Node(l, y, r)) = Node(insert(y, l), x, r)

(* trav : 'a tree -> 'a list *)
fun trav Empty = [ ]
  | trav (Node(l, x, r)) = (trav l) @ (x :: trav r)

(* combine : 'a tree * 'a tree -> 'a tree *)
fun combine (A, B) = foldr insert B (trav A)
```

For all types `t` and all values `A` and `B` of type `t tree`, `combine(A, B)` returns a tree value consisting of all of the items from `A` and all of the items from `B`. Using this function we can define a tree-filtering function.

```
(* filtertree : ('a -> bool) -> ('a tree -> 'a tree) *)
(* REQUIRES: p is a total function                                    *)
(* ENSURES: filtertree p T ==> a tree consisting of all items from T *)
(*          that satisfy p, in the same order as in T.                *)

fun filtertree p Empty = Empty
  | filtertree p (Node(l, x, r)) =
     if (p x) then Node(filtertree p l, x, filtertree p r)
              else combine(filtertree p l, filtertree p r)
```

Transforming the data in a tree

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

A function for adding an integer to the integers in an `int tree`:

```
fun addtonodes(x:int): int tree -> int tree  =  treemap (fn y => x+y)
```

Combining the data in a tree

If we have a `t tree`, a base value `z:t`, and a combining function `g:t*t->t`, we can combine all the data in the tree with `z` using a "treefold". Here is one way to do this, encapsulated as a higher-order function:

```
(* treefold : ('a * 'a -> 'a) -> 'a -> 'a tree -> 'a *)
fun treefold g z Empty = z
  | treefold g z (Node (l, x, r)) = g(x, g(treefold g z l, treefold g z r))
```

Actually, this is not the only possible way to combine the data. We could have used instead the function

```
(* treefold' : ('a * 'a -> 'a) -> 'a -> 'a tree -> 'a *)
fun treefold' g z Empty = z
  | treefold' g z (Node (l, x, r)) = g(treefold' g z l, g(x, treefold' g z r))
```

and there are other alternatives. Sometimes when we use folding on trees we have an *associative* and *commutative* operation `g`, and the "base value" `z` is a "zero" for `g`, so that `g(x, z) = x` for all values `x`. In such cases we get the same combined result, no matter what order we combine the data. Otherwise we may need to pay attention to the order in which the combinations happen!

Using `treefold` we can obtain a function for adding the integers in an integer tree:

```
(* sum_tree : int tree -> int *)
val sum_tree = treefold (op +) 0
```

Similarly, a function for adding all the integers in a tree of integer trees(!):

```
fun count_tree T = sum_tree (treemap sum_tree T)
```

Contrast this with the analogous function for adding the integers in a list of integer lists from page 13.

Doing two things at once

Often we have a collection of data in a tree and we need to apply some function to all of the data items and then combine the results. We could do this by applying a `treemap` followed by a `treefold`. Notice that we would be constructing, as an intermediate data structure, the entire tree produced by the map phase, merely to "consume" this tree in the fold phase. We may wish to skip that step by performing the mapping and folding in one fell swoop. Here is one possible function designed to encapsulate this situation:

```
(* mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a tree -> 'b *)
fun mapreduce f z g Empty = z
  | mapreduce f z g (Node(l, x, r)) =
          g (f x, g(mapreduce f z g l, mapreduce f z g r))
```

As intended, `mapreduce` does behave applicatively like a `treemap` followed by a `treefold`. We can prove that for all suitably typed, total functions `f` and `g`, and all suitably typed values `z` and `T`,

`mapreduce f z g T` $\cong$ `treefold g z (treemap f T).`

Contrast this with the following function that actually does the treemap and then the treefold:

```
fun mapreduce' f z g t  =  treefold g z (treemap f t)
```

A tree folding function with three arguments

For binary trees with data at the nodes, it is natural to fold a three-argument function over the tree. We then obtain a treefolding function like this:

```
(* treeFold : ('a * 'b * 'a -> 'a) -> 'a -> 'b tree -> 'a *)
fun treeFold g z Empty = z
  | treeFold g z (Node (l, x, r)) = g (treeFold g z l, x, treeFold g z r)
```

We may now implement a function to sum all the elements in an `int tree` as follows:

```
(* sumTree : int tree -> int *)
val sumTree = treeFold (fn (a,x,b) => a + x + b) 0
```