# 15–150: Principles of Functional Programming

## *More about Higher-Order Functions*

Michael Erdmann*
Spring 2023

## 1 Topics

- Currying and uncurrying
- Staging computation
- Partial evaluation
- Combinators

## 2 Currying and uncurrying

We've already seen that a function of several arguments (whose argument type is a tuple type) has a curried version, a function that takes a single argument (the first component in the tuple) and returns a function of the remaining arguments.

For example, the ordinary (uncurried) addition function on the integers is `add` below, and `add'` is the curried version. Either of the two definitions for `add'` is acceptable SML syntax. The first one emphasizes the fact that `add' x` returns a function explicitly. The second definition uses a curried format that reflects the fact that we can apply `add'` to an integer (which returns a function) and then apply (this function) to another integer, as in `add' 3 4`. The SML convention that application associates to the left ensures that this expression is parsed as `(add' 3) 4`.

*Adapted from documents by Stephen Brookes and Dan Licata.

```
(* add : int * int -> int *)
fun add (x:int, y:int) : int = x+y

(* add' : int -> (int -> int) *)
fun add' (x:int) : int -> int = fn y:int => x+y

(* add' : int -> (int -> int) *)
fun add' (x:int) (y:int) : int = x+y
```

The SML convention that the arrow type constructor associates to the right also ensures that `int -> int -> int` is parsed as `int -> (int -> int)`, so the parentheses in the above type comments for `add'` are redundant.

Note that in the scope of the above definitions, the value of `add` is a function value of type `int * int -> int`, and the value of `add'` is a function value of type `int -> (int -> int)`. These are *not* the *same* value — they don't even have the same type!

```
add   ≅  fn (x:int, y:int) => x+y
add'  ≅  fn x:int => (fn y:int => x+y)
```

Of course we can easily generalize, and if we have an uncurried function definition it's easy to make a syntactic transformation and obtain a curried function definition that corresponds to it, in the same way that `add'` corresponds to `add`.

We can say more precisely what we mean by "correspondence" between an uncurried function and a curried function. Let `t1, t2,` and `t` be types. The (uncurried) function `f:t1*t2 -> t` corresponds to the (curried) function `g:t1 -> (t2 -> t)` if and only if for all values `v1:t1` and `v2:t2`,

```
f(v1, v2) ≅ (g v1) v2
```

Again, because application associates to the left, this is the same as saying: for all values `v1:t1` and `v2:t2`,

```
f(v1, v2) ≅ g v1 v2
```

If the result type `t` is a ground type[1], this means that for all values `v1:t1` and `v2:t2`, either `f(v1, v2)` and `g v1 v2` both loop forever, or both raise the same exception, or both evaluate to the same value of type `t`.
If the result type `t` is not a ground type, equivalence for values of type `t` isn't "being the same value", but instead is based on extensionality. Specifically,

[1]Here "ground type" means a type built from base types like `int`, `real`, `bool` without using arrows.

in this case the requirement is that for all values v1:t1 and v2:t2, either f(v1, v2) and g v1 v2 both loop forever, or both raise the same exception, or both evaluate to values and the values are extensionally equivalent.

One may verify that, according to this definition, add and add' do indeed correspond as required. Note that values of type int*int are pairs (m,n) where m and n are values of type int. So for all values m and n of type int, we have

```
add(m,n)  ==> (fn (x:int, y:int) => x+y)(m,n)
          ==> [m/x,n/y] (x+y)
          ==> m+n
          ==> the SML value for m+n

(add' m) n ==> ((fn x:int => (fn y:int => x+y)) m) n
           ==> ([m/x] (fn y:int => x+y)) n
           ==> [m/x][n/y]x+y
           ==> the SML value for m+n
```

Thus we have indeed shown that add and add' correspond.

Evaluation of an application expression e1 e2 first evaluates e1 to a function value (say f), then evaluates e2 to a value (say v), then evaluates f(v), as follows. If f is a clausal function, say fn p1 => e1' | ...| pk => ek', we find the first clause whose pattern pi matches v successfully, then plug in the value bindings produced by this match into the right-hand-side expression ei'.

In the previous derivation we used notation (introduced earlier in the semester) such as [m/x, n/y] for the list of value bindings produced by successfully matching the pattern (x:int, y:int) with the value (m,n). And we wrote

```
[m/x, n/y] (x+y)
```

to indicate an evaluation of the expression (x+y) in an environment with the bindings [m/x, n/y]. This evaluation amounts to substituting the values m and n for the free occurrences of x and y in the expression x+y. In this example, this produces the expression m+n.

Similarly, matching the pattern x:int with value m produces the value binding [m/x]. Subsequently, matching the pattern y:int with value n produces the value binding [n/y].

Make sure you understand how the derivations above can be justified by appealing to the function definitions for add and add' and using the evaluation strategy described in the previous paragraph.

## 3 Currying as a higher-order function

The idea of currying is very generally applicable. In fact we can encapsulate this idea as an SML function

```
curry : ('a * 'b -> 'c) -> ('a -> ('b -> 'c))
```

whose polymorphic type indicates its general utility. Again, since arrow associates to the right we can omit some of the parentheses:

```
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Again there are alternative syntactic ways to write this function in SML:

```
fun curry f = fn x => fn y => f(x,y)
fun curry f x = fn y => f(x,y)
fun curry f x y = f(x,y)
```

We could also include type annotations, as in

```
fun curry (f:'a*'b -> 'c) = fn x:'a => fn y:'b => f(x,y)
```

Given an uncurried function, we can either produce by hand a curried version by following the recipe used for `add` and `add'`, or we can simply apply `curry` to the uncurried function.

For illustration, consider the Ackermann function as defined by:

```
(* ack : int * int -> int *)
fun ack (x:int, y:int) : int =
      (case (x,y) of
           (0, _) => y+1
         | (_, 0) => ack (x-1, 1)
         | (_, _) => ack (x-1, ack(x, y-1)))
```

Here is a curried version:

```
(* ack' : int -> int -> int *)
fun ack' (x:int) : int -> int =
  (fn (y:int) =>
    (case (x,y) of
         (0, _) => y+1
       | (_, 0) => ack' (x-1) 1
       | (_, _) => ack' (x-1) (ack' x (y-1))))
```

We had to be careful in deriving the definition of `ack'` from the definition of `ack`; recursive calls in `ack` of form `ack(a,b)` get transformed into recursive calls in `ack'` of form `ack' a b`.

There are also many other syntactic ways to define a curried version of `ack`. Here is the easiest way to obtain a curried version:

```
(* curried_ack : int -> int -> int  *)
val curried_ack = curry ack
```

The functions `ack` and `ack'` correspond, in that for all values `m` and `n` of type `int`,

`ack(m, n)` $\cong$ `ack' m n`

The functions `ack'` and `curried_ack` are extensionally equivalent, i.e., for all values `m` of type `int`,

`ack' m` $\cong$ `curried_ack m`

Since these expressions have type `int -> int`, and both evaluate to a value, this is the same as saying that: for all values `m` and `n` of type `int`,

`ack' m n` $\cong$ `curried_ack m n`

How would you go about the task of *proving* the assertions made above about these functions?

- The fact that `ack` and `curry ack` correspond follows from the definition of `curry` and the definition of equivalence at the relevant types. In fact it, one can show that for *all* types `t1`, `t2`, and `t`, and *all* function values `f:t1*t2->t` and all values `v1:t1` and `v2:t2`,
  `f(v1, v2)` $\cong$ `(curry f) v1 v2`.

[Aside: Historically, the Ackermann function is interesting because it is well known as a *general recursive* function (on the non-negative integers) that grows faster than any *primitive recursive* function (and hence is not itself expressible using primitive recursion). (The distinction between "general recursive" and "primitive recursive" has to do with the way the function definition is constructed, and may be familiar to you if you have studied "recursive function theory" and/or the theory of computability.)]

## 4 Exponentiation

Consider the function

```
exp : int*int -> int
```

for computing the value of $\mathtt{b}^{\mathtt{e}}$ when `e` is a non-negative integer ("exponent") and `b` is an integer ("base"):

```
fun exp (e, b) =
   if e=0 then 1 else
        b * exp (e-1, b)
```

For all integer values $\mathtt{e} \geq 0$ and `b`, `exp(e, b)` evaluates to the value of $\mathtt{b}^{\mathtt{e}}$. [Aside: we use an `if-then-else` expression rather than function clauses, in order to focus on and simplify the discussion of currying.]

Since exponentiation with a particular base is a widely used operation it would be helpful to take advantage of currying, and use:

```
(* curried_exp : int -> (int -> int) *)
fun curried_exp e b =
    if e=0 then 1 else
         b * curried_exp (e-1) b
```

or (just as acceptable)

```
(* curried_exp : int -> (int -> int) *)
fun curried_exp e =
    fn b => if e=0 then 1 else
                 b * curried_exp (e-1) b
```

or even

```
(* curried_exp : int -> (int -> int) *)
fun curried_exp e =
    if e=0 then fn _ => 1 else
         fn b => b * curried_exp (e-1) b
```

(Figure out why these last two function definitions are equivalent!)

For example, we might then define:

```
(* one, id, square, cube : int -> int *)
val one = curried_exp 0
val id = curried_exp 1
val square = curried_exp 2
val cube = curried_exp 3
```

In the scope of these definitions, `one` behaves like the constant function that always returns 1, `id` behaves like the identity function on integers, `square` behaves like the squaring function on integers, and `cube` behaves like the cubing function on integers.

In terms of evaluation-with-substitution, we get the following equivalences:

```
one ≅ curried_exp 0
    ≅ (fn e => (fn b => if e=0 then 1 else b * curried_exp (e-1) b)) 0
    ≅ [0/e] (fn b => if e=0 then 1 else b * curried_exp (e-1) b)
    ≅ (fn b => if 0=0 then 1 else b * curried_exp (0-1) b).
```

Consequently, for all values `b` of type `int`,

```
one b ≅ 1
```

This shows that `one ≅ (fn b:int => 1)`. We emphasize that it is *not* the case that `one ==> (fn b:int => 1)`.

Similarly, we get

```
id ≅ curried_exp 1
   ≅ (fn e => (fn b => if e=0 then 1 else b * curried_exp (e-1) b)) 1
   ≅ [1/e] (fn b => if e=0 then 1 else b * curried_exp (e-1) b)
   ≅ (fn b => if 1=0 then 1 else b * curried_exp (1-1) b)
   ≅ (fn b:int => b).
```

Exercise: Perform a similar analysis for `square` and `cube`. Show that

```
square ≅ (fn b:int => b*b)
cube ≅ (fn b:int => b*b*b)
```

**Comment:** Above, we showed that

```
one ≅ (fn b => if 0=0 then 1 else b * curried_exp (0-1) b)
one ≅ (fn b => 1)
```

The two function values appearing here are *not the same values.* But they are *extensionally* equivalent. Consider the function `(fn b => if 0=0 then 1 else b * curried_exp (0-1) b)`. Every time we apply this function value to an argument value, the expression `0=0` gets evaluated (it evaluates to `true`, of course) and the result of the application is then found to be `1`.

In contrast, when we apply `(fn b => 1)` to an argument value, there is no check of the form `0=0`, so the result `1` is computed more quickly (albeit imperceptibly more quickly in this example).

In more realistic examples, the additional work done inside the function body may be significant, and it may be more efficient to pre-compute some expression value(s). This is the main idea behind *staging computation.*

# 5 Staging

When you make pancakes, you start by mixing the dry ingredients (flour, sugar, baking powder, salt), then mix in the wet ingredients (oil, eggs, milk). Even if you only have the dry ingredients (maybe your breakfast partner has just gone to the grocery store to buy eggs and milk), you can do useful work: mixing the dry stuff. That should save you a few valuable seconds when the rest of the ingredients arrive. The same idea is also relevant for programming. A *multi-staged* function does useful work when "partially" applied. Applying a curried function to its first argument *specializes* the function and generates code specific to that first argument. This can improve efficiency when the specialized function is used many times. *Staging* is the programming technique of writing multi-staged functions.

One application of staging is to reduce the evaluation overhead. For example, we can write a staged exponentiation function that doesn't recur on exponent value every time it is called. The idea is to delay asking for the base value until we have entirely processed the exponent value and produced a specialized function that will behave like "exponentiation with the given exponent".

Here is a staged version of exponentiation.

```
(* staged_exp : int -> int -> int *)
fun staged_exp e =
  if e=0 then fn _ => 1 else
     let
        val oneless = staged_exp (e-1)
     in
        fn b => b * oneless b
     end
```

(Compare this carefully with the various ways we gave for defining `curried_exp` and make sure you understand the difference!)

For all values `e` $\geq 0$, `staged_exp e` evaluates to a function value that is extensionally equivalent to `fn b => exp(e, b)`. (And actually most compilers will optimize to take account of the value of `e`.) Even without optimization we get

```
   staged_exp 0 ==> (fn _ => 1)
```

and

```
staged_exp 1
  ==> (fn e => if e=0 then . . . else let . . . ) 1
  ==> if 1=0 then . . . else let val oneless = . . .
  ==> let val oneless = staged_exp 0 in fn b => b * oneless b end
  ==> let val oneless = (fn _ => 1) in fn b => b * oneless b end
  ==> [(fn _ => 1)/oneless] (fn b => b * oneless b)
```

There is hardly any evaluational overhead left! When we apply this function value to an argument value only a small amount of work gets done. A smart compiler might even optimize[2] this function value, first substituting the binding `[(fn _ => 1)/oneless]` into the lambda expression to obtain `(fn b => b * ((fn _ => 1) b))`, then optimizing to `(fn b => b * 1)`, and finally to `(fn b => b)`.

Contrast this with the evaluational behavior of `curried_exp`:

```
curried_exp 1
   ==> (fn e => (fn b => if e=0 then 1 else b * curried_exp (e-1) b)) 1
   ==> [1/e] (fn b => if e=0 then 1 else b * curried_exp (e-1) b)
```

Here there's still some evaluational overhead: Although a smart compiler might optimize to `(fn b => b * curried_exp 0 b)`, that still leaves a recursive call to `curried_exp` lurking inside the function body. Even in this simple example, the staged version of exponentiation yields a function value that operates more efficiently when called than does the merely curried version.

See the lecture code for another, simpler, example in which staging is beneficial.

[2]In principle, a compiler is free to use any equivalences as code optimizations, to replace a value by any other equivalent value, without changing the evaluational behavior of your program. So why wouldn't you want a compiler to be capable of transforming the original `exp` function into this form as well? One reason: it's good to have a *predictable* cost model, and letting the compiler do arbitrary things makes it hard to predict performance. And secondly, optimizations that require expanding recursive calls are tricky to apply, because there is a termination worry: when do you stop optimizing? Additionally, there's a tradeoff, because by unrolling the recursion you increase the size of the code. The nice thing about staging is that it lets you express a desired optimization yourself, modulo some harmless steps that can safely be left to the compiler.

## 6 Combinators

Another benefit of higher-order functions is that one may "lift" operations from some type to functions that map into that type.

For instance, if `f` and `g` are two functions mapping into the integers, say of type `t -> int` for some type `t`, then we may combine `f` and `g` as if they were themselves integers.

As a first example, let's define a higher-order function `++` that adds `f` and `g` to produce a new function `f ++ g` of type `t -> int`. The higher-order function `++` is an instance of what is sometimes called a *combinator*.

How does one add integer-valued functions? Using the *pointwise principle* from math: `f ++ g` is the function whose value at a given "point" `x` is `f(x) + g(x)`. In SML we may express this as:

```
infixr ++
fun (f ++ g) (x : 'a) : int = f(x) + g(x)
```

If we have these declarations:

```
fun square (x:int):int = x*x
fun twice (x:int):int = 2*x,
```

then `square` represents the math function $x^2$ and `twice` represents the math function $2x$. The following declaration

```
val quadratic = square ++ twice
```

would therefore produce an SML function called `quadratic` that represents the math function $x^2 + 2x$.

We used the combinator `++` to define the function `quadratic` without needing to refer to its argument `x`. That approach is sometimes called *point-free programming*. Curried higher-order functions frequently facilitate point-free programming.

As another example, suppose we define the combinator `MIN` by

```
fun MIN (f, g) (x : 'a) : int = Int.min(f(x), g(x))
```

Now consider

```
val lowest = MIN(square, twice)
```

The SML function `lowest` represents the lower envelope of the two functions $x^2$ and $2x$. (Graph these functions and you will see!)