# 15-150 Fall 2020



Lecture 12

Using higher-order functions

# 1 Introduction

We develop a series of higher-order functions to solve a problem about making change, a variation on the well known subset-sum problem. Given a list of coins (positive integers) and a monetary amount (a non-negative integer) we want to find if it is possible to make exact change for the amount by using a sublist of the coins, in such a way that the sublist of coins satisfies some constraint. The constraint is represented as a (total) function from integer lists to truth values. For example, we might want a sublist of coins consisting of exactly three coins. This could be represented as the function

```
fn L:int list => (length L = 3)
```

for example, and I'm sure you can think up some other non-trivial constraints and represent them as ML functions like this. Another one of relevance is the "always true" constraint `fn L:int list => true`.

We begin with a simple but horribly inefficient solution, and progress to a more efficient solution, leading to a more general function that solves a more general problem. At each stage in the development we pay careful attention to types and specifications.

The same problems and code fragments are typically discussed in class, with some audience participation (but this is difficult to achieve when using zoom!). Here we offer the code again, with partial commentary to remind you of the key points, and we give more technical proof details concerning the correctness of our functions.

As mentioned above any constraint function `p : int list -> bool` that we are likely to want to use is *total*, i.e. we always get a definite answer (`true` or `false`) when we ask if a given coin list satisfies the constraint. We'll build this assumption into the specifications.

## 2 A bad solution

Obviously we need a function that can be applied to an integer amount, a list of coin denominations, and a constraint function, to return a boolean value. It's not necessary to insist on currying, or to group the arguments together as a tuple in a particular order, so we'll just make a decision and stick to it. Since our first attempt at this problem will turn out to have undesirable features, we'll give it a pejorative name.

So we'll adopt the following specification:

- `badchange : int * int list -> (int list -> bool) -> bool`
- REQUIRES `n` $\geq 0$, `L` a list of positive integers, `p` a total function
- ENSURES `badchange (n, L) p` = `true` if there is a sublist `A` of `L` such that `sum A = n` and `p A = true`; `badchange (n, L) p` evaluates to `false` otherwise.

(We could just as well have said "evaluates to" rather than "=" here, as it means the same thing for expressions of type `bool` or `int`.)

To solve this problem we'll use the following helper functions:

```
fun sublists [ ] = [ [ ] ]
|   sublists (x::R) =
     let
        val S = sublists R
     in
        S @ map (fn L => x::L) S
     end

fun exists p [ ] = false
|   exists p (x::R) = p(x) orelse exists p R

fun sum L = foldr (op +) 0 L
```

Note that `sublists L` evaluates to a list of all the sublists of `L`, with `L` itself as the final sublist. And `exists p L` evaluates to `true` if there is an element of `L` that satisfies `p`, and evaluates to `false` otherwise.

Using these helpers, the following definition seems "obvious":

```
fun badchange (n, L) p =
        exists (fn A => sum A = n andalso p A) (sublists L)
```

It is possible to prove, by induction on `L`, that this function does indeed satisfy the specification. We omit the details, since we will show instead that the function is horribly inefficient.

Example:

- `badchange(300,[1,2,3,...,24])(fn _ => true)` takes a very long time to produce the answer `true`
- Reason: builds the list of all sublists (there are $2^{24}$ of them), checks them sequentially, and only the final one has the desired sum.

Incidentally, similar inefficiency will also occur (in worst case) if we try to be slightly less "brute force" by only checking the predicate `p` on the sublists of `L` that add up to `n`, or only check the sum of sublists satisfying `p`. Consider, for example:

```
fun slowchange(n, L) p =
   let
     val R = List.filter p (sublists L)
   in
     exists (fn A => (sum A = n)) R
   end
```

This still computes the entire list of all sublists! Even if `p` is a constant-time function (unlikely, since its argument is a list) the work for `slowchange(n, L) p` is going to be (more than) exponential in the length of `L`. (Even the `sum` calls incur some non-trivial costs, and since we already know that the function is so slow it's not really worth figuring out the work accurately.) An alternative, first filterig for sublists that have the correct sum, would be:

```
fun slowchange(n, L) p =
   let
     val R = List.filter (fn A => (sum A = n)) (sublists L)
   in
     exists p R
   end
```

but again this computes the entire list of sublists so will be painfully slow for large `n`.

# 3 A better solution

Instead let's design a function with a different name but the same type and specification, that:

- Doesn't build the entire list of sublists.
- Only calls `p` on sublists with the correct sum.
- Returns the result immediately on special cases `n = 0` (zero amount), `L = [ ]` (no coins) where the answer is obvious.
- Uses *recursion* when `n>0` and `L=x::R`, with a shorter coin list. (There's an easy way to do this: try using the first coin in the list, and if that doesn't work try without using the first coin.)

Specification:

- `change : int * int list -> (int list -> bool) -> bool`
- REQUIRES `n` $\geq 0$, `L` a list of positive integers, `p` a total function
- ENSURES `change (n, L) p` evaluates to `true` if there is a sublist `A` of `L` such that `sum A = n` and `p A = true`; evaluates to `false` otherwise.

The `change` function definition is designed by paying attention to the spec: in each clause we assume that the function's arguments satisfy the requirements and we pick a right-hand-side expression that will ensure the correct result, provided any recursive calls that we make work correctly. We must be careful only to make recursive calls on arguments that also obey the requirements, so that we can rely on their results satisfying the spec. (If we do this, it should be straightforward to prove by induction that the entire function satisfies the spec!)

```
fun change (0, L) p = p [ ]
|   change (n, [ ]) p = false
|   change (n, x::R) p =
        if x <= n
        then change (n-x, R) (fn A => p(x::A))
                orelse
             change (n, R) p
        else change (n, R) p
```

- Style note: see how easy it is to build a suitable constraint for the recursive call in the then-branch, using ML function syntax.
- Type check: verify that `change` has the intended type.
- Requirements check: verify that whenever `(n,L,p)` satisfy the REQUIRES properties and `change(n, L) p` makes a recursive call, the parameters in the recursive call also satisfy the REQUIRES properties. In particular, note that when `x <= n` is `true`, `n - x` has a non-negative value. And when `x` is an integer value and `p` is a total function of type `int list -> bool`, so is `fn A => p(x::A)`.
- Termination check: find a non-negative quantity that decreases in each recursive call, assuming that the initial arguments to the function satisfy the REQUIRES property.
- Make sure you see why we use `p [ ]` when the amount is zero. Try to convince yourself that the if-then-else code is written in accordance with the specification, so that *assuming* the recursive calls obey the spec, it follows that the if-then-else combination produces the correct answer for `change (n, x::R) p`.
- Try some examples.

## Correctness

Next we show in a bit more detail that `change` satisfies its specification. Let $P$(`L`) be the following property:

> For all `n` $\geq 0$, and all total functions `p` of type `int list -> bool`
>
> (a) `change (n, L) p = true` if there is a sublist `A` of `L` such that `sum A = n` and `p A = true`;
>
> (b) `change (n, L) p = false` otherwise.

We will prove that:

> For all lists `L` of positive integers, $P$(`L`) holds.

This is what we mean by saying that `change` satifies its specification, or (more briefly) `change` is *correct*.

The `change` function makes no recursive calls when `n = 0` or `L = [ ]`, so we will treat these as base cases in the induction proof. Otherwise, when `n` $> 0$ and `L` is non-empty, the function makes (one or more) recursive calls in which the list parameter is the tail of `L`. So we will use structural induction on lists to handle these cases. (We could just as easily use induction on the length of the coin list.)

Before we begin the proof, we state the *value equations* that hold for this function, given the definition above. These hold, for all values `n` and `x` of type `int`, `L` and `R` of type `int list`, and `p` of type `int list -> bool`:

(i) `change (0, L) p = p [ ]`

(ii) `change (n, [ ]) p = false`
if `n > 0`

(iii) `change (n, x::R) p =`
`change(n-x, R)(fn A => p(x::A)) orelse change(n, R)p`
if `n` $> 0$ and `x` $\leq$ `n`

(iv) `change (n, x::R) p = change (n, R) p`
if `n` $> 0$ and `x` $>$ `n`.

Because of the way `orelse` works, we can rewrite (iii) as:

(iii-a) When `n` $> 0$ and `x` $\leq$ `n`, `change (n, x::R) p = true` if

either `change (n-x, R) (fn A => p(x::A)) = true`

or (`change (n-x, R) (fn A => p(x::A)) = false`
and `change (n,R) p = true`)

(iii-b) When `n` $> 0$ and `x` $\leq$ `n`, `change (n, x::R) p = false` if

`change (n-x, R) (fn A => p(x::A)) = false`
and `change (n,R) p = false`

We also note the following simple facts about sublists:

1. The only sublist of `[ ]` is `[ ]`.
2. `A` is a sublist of `x::R` iff either `A` is `x::B` with `B` a sublist of `R`, or `A` is a sublist of `R`.

## Theorem

For all lists `L` of positive integers, $P$(`L`) holds.

## Proof

By structural induction on `L`.

- For `n = 0` we know by (i) that for all values `L` and `p`,

  `change (0, L) p = p [ ]`.

  Suppose `p` is total. The only sublist of `L` with sum equal to 0 is `[ ]`. And since `p` is total, `p [ ]` is equal to a truth value, so

  `change (0, L) p = true if p [ ] = true,`

  `change (0, L) p = false if p [ ] = false`.

  So `change (0, L) p = true` if there is a sublist `A` of `L` with sum equal to 0 such that `p A = true`, and `change (0, L) p = false` otherwise.

- For `n` $> 0$ and `L = [ ]`, by (ii) we have

  `change (n, [ ]) p = false`.

  There is no sublist of `[ ]` with sum equal to `n` because `n` is not 0. So `false` is the correct result as specified for this case.

- Inductive case: Assume as induction hypothesis that $P$(`R`) holds, i.e. For all `m` $\geq 0$, and all total functions `q` of type `int list -> bool`

  (a) `change (m, R) q = true` if there is a sublist `A` of `R` such that `sum A = m` and `q A = true`;

  (b) `change (m, R) q = false` otherwise.

  Let `L = x::R` be a list of positive integers, `n` $> 0$, and let `p` be a total function of type `int list -> bool`. We must show that

  (i) `change (n, x::R) p = true` if there is a sublist `A` of `x::R` such that `sum A = n` and `p A = true`;

(ii) `change (n, x::R) p = false` otherwise.

If `x` $\leq$ `n` we have equations (iii-a) and (iii-b) as above. Since `p` is total and `x` is an integer value, `fn A => p(x::A)` is also a total function value of type `int list -> bool`. So by the induction hypothesis $P$(`R`),

> `change (n-x, R) (fn A => x::A) = true` if there is a sublist `B` of `R` such that `sum B = n-x` and `(fn A => p(x::A)) B = true`;
>
> `change (n-x, R) (fn A => p(x::A)) = false` otherwise.

In other words:

> `change (n-x, R) (fn A => x::A) = true` if there is a sublist `B` of `R` such that `sum B = n-x` and `p(x::B) = true`;
>
> `change (n-x, R) (fn A => p(x::A)) = false` otherwise.

Clearly when `sum B = n-x` we get `sum (x::B) = n`, because `x+(n-x)=n`.

Also by induction hypothesis $P$(`R`) we have:

> `change (n, R) p = true` if there is a sublist `A` of `R` such that `sum A = n` and `p A = true`;
>
> `change (n, R) p = false` otherwise.

Using facts about sublists given earlier, and (iii-a) and (iii-b), we get [1]

> `change (n, x::R) p = true` if there is a sublist `A` of `x::R` such that `sum A = n` and `p A = true`;
>
> `change (n, x::R) p = false` otherwise.

as required.

If `x` $>$ `n` we have equation (iv) above. Using induction hypothesis $P$(`R`) again we can show that in this case we also get

> `change (n, x::R) p = true` if there is a sublist `A` of `x::R` such that `sum A = n` and `p A = true`;

[1] There are details needed here to make the justification clearer, but we leave them to the reader. All the essential ingredients are in place, using the given assumptions and basic facts about sublists.

`change (n, x::R) p = false` otherwise.

as needed. [2]

# 4 Assessment

So now we have a function `change` of type

```
int * int list -> (int list -> bool) -> bool
```

that satisfies the given specification. It can be used to check if there is a "suitable" subset of `L` that adds up to `n`. Calling the function with an appropriate combination of arguments will give us an answer in the form of `true` or `false`. And we know from the spec what this truth value means. If the result is `true`, there must be a sublist of coins that "works", and if `false` there's no suitable sublist. Unfortunately we also have a problem caused by "boolean blindness". If we actually wanted to know *which* sublist is "suitable", if there was one, the `change` function won't tell us. We designed it to only calculate a truth value. By focussing on boolean results we threw away the ability to produce more information! In order to get more information out we'll need to design a function with a more complex result type. In this case it seems natural to ask for a result of type `int list option`. We would interpret a result value of form `SOME A` as meaning "`A` is a suitable sublist", and the result `NONE` as meaning "there is no suitable sublist".

# 5 A more general solution

We now offer a function `mkchange` of type

```
int * int list -> (int list -> int list option) -> int list option
```

with the specification Specification:

- REQUIRES `n` $\geq 0$, `L` a list of positive integers, `p` a total function

[2] Again, we omit the details. In particular, since `x` $>$ `n` the only sublists of `x::R` with sum equal to `n` are also sublists of `R`.

- ENSURES `mkchange (n, L) p = SOME A` where `A` is a sublist of `L` such that `sum A = n` and `p A = true`, if there is one; is equal to `NONE` otherwise.

The control flow for `mkchange` is similar to `change`. The main differences concern the values produced as results: we need to return an optional list, and we need to use pattern-matching on values of type `int list option` rather than on boolean values.

```
fun mkchange (0, L) p = if p [ ] then SOME [ ] else NONE
|   mkchange (n, [ ]) p = NONE
|   mkchange (n, x::R) p =
         if x <= n
         then case mkchange (n-x, R) (fn A => p(x::A)) of
                    SOME B => SOME (x::B)
                  | NONE   => mkchange (n, R) p
         else mkchange (n, R) p
```

The proof that this function satisfies its spec is very similar to the proof given earlier for `change`. Not surprising, since the two functions have similar control flow! Instead of writing out the adapted proof, we'll leave that as an exercise and instead look at an example.

### An example

Let `L` be `[10,5,5]` and `p` be `fn A => length(A) > 1`. We expect that `mkchange (10,L) p` is equal to `SOME [5,5]`.

The value equations, based on the definition of `mkchange` are as follows. For all values `n` and `x` of type `int`, `L` and `R` of type `int list`, and `p` of type `int list -> bool`:

```
(1)   mkchange (0, L) p = if p [ ] then SOME [ ] else NONE
(2)   mkchange (n, [ ]) p = NONE
           for n<>0
(3a)  mkchange (n, x::R) p =
         case mkchange (n-x, R) (fn A => p(x::A)) of
               SOME B => SOME (x::B)
            |  NONE   => mkchange (n, R) p
          for n>0 and x <= n
```

```
(3b)  mkchange (n, x::R) p = mkchange (n, R) p
          for n>0 and x>n
```

So we have (using equation (3a)),

```
   mkchange (10, [10,5,5]) p
= case mkchange (10-10, [5,5]) (fn A => p(10::A)) of
              SOME B => SOME (10::B)
          |   NONE   => mkchange (10, [5,5]) p
```

But `10 - 10 = 0` and `fn A => p(10::A)` is a value, so we can use referential transparency, value equation (1), and the definition of `p`, to show that

```
   mkchange (10-10, [5,5]) (fn A => p(10::A))
 = mkchange (0, [5,5]) (fn A => p(10::A))
 = if (fn A => p(10::A)) [ ] then SOME [ ] else NONE
 = if p [10] then SOME [ ] else NONE
 = if (length [10] > 1) then SOME [ ] else NONE
 = NONE
```

Hence, continuing from above we get

```
   mkchange (10, [10,5,5]) p
= case mkchange (10-10, [5,5]) (fn A => p(10::A)) of
              SOME B => SOME (10::B)
          |   NONE   => mkchange (10, [5,5]) p
= case NONE of
              SOME B => SOME (10::B)
          |   NONE   => mkchange (10, [5,5]) p
= mkchange (10, [5,5]) p
```

By a similar analysis we can show that

```
mkchange (10, [5,5]) p = SOME [5,5],
```

so we have, making `p` explicit,

```
mkchange (10, [10,5,5]) (fn A => length A > 1) = SOME [5,5],
```

as expected.

# 6 Assessment, revisited

Now we have two functions: `change`, of type

```
int * int list -> (int list -> bool) -> bool
```

and `mkchange`, of type

```
int * int list -> (int list -> bool) -> int list option.
```

We've argued that `mkchange` is more informative, in that when applied to a given combination of `n, L, p` its result conveys more information than we would get by applying `change` to the same arguments. In fact we can make this claim more precise as follows: `mkchange` can be used to define a function that satisfies the specification we gave earlier for `change`.

```
fun change (n, L) p = case mkchange (n, L) p of
                        SOME _ => true
                      | NONE   => false
```

We're not saying this is a *good* way to write the two functions, just that it is a true assertion about their behavior.

The developments so far may already suggest to you that there are bigger issues lurking in the background. Indeed, what's to prevent someone suggesting that we build yet another "more flexible" change-finding function with an even richer result type? How general and flexible can we get? The answer involves polymorphism (and introduces "continuations").

# 7 A polymorphic solution

We can design a more flexible function by assuming that there is some type (as yet unspecified, to be chosen later) of "answers", and augmenting the type of the change function with additional arguments (two functions) that play very specific rôles:

- a function `s` from `int list` to answers, that says how to convert a "suitable integer list" into an answer, if you find one;
- a function `k` from the type `unit` to answers, that says what to do to produce an answer if you discover there is no suitable integer list.

We refer `s` as a "success continuation", and `k` as a "failure continuation". In the new, more flexible specification for the more flexible function, we describe how to use `s` and `k`. The (most general) type of this new function is polymorphic, with a type variable standing for the type of answers. Thus we will be able to instantiate this type by choosing a specific answer type (such as `bool`, `int list option`, or whatever). And we will be able to use this function at any instance of its (most general) type. In particular we will be able to obtain functions that behave like the original `change` and `mkchange` in this manner!

The new function will be

```
changer : int * int list -> (int list -> bool)
                            -> (int list -> 'a) -> (unit -> 'a) -> 'a
```

and its specification will be

```
REQUIRES  n>=0, L a list of positive integers, p total

ENSURES  changer (n, L) p s k = s A
                        where A is a sublist of L such that
                              sum A = n and p A = true,
                                 if there is one

         changer (n, L) p s k = k ( )
                                 otherwise
```

So `changer (n, L) p s k` either calls `s` with a "suitable" sublist of `L`, if there is one, or calls `k( )` if there isn't one. Note that we don't say *anything*

about what `s` and `k` do! The spec says that the `changer` function does essentially the same thing regardless of what the continuations are! This is what we mean by being very flexible.

Here is the definition of `changer`. Note the very familiar control flow concerning the pattern-matching on lists and so on. However, since we don't know what the type of answers is, and the recursive calls don't "return" a value but merely pass some value to a continuation, the function body cannot do pattern-matching on booleans or options. Instead we use the continuations to say what should happen in case of success or failure. It's easy to write code for a continuation – just use `fn` and write what's appropriate for the continuation to "do next if called".

```
fun changer (0, L) p s k =
             if p [ ] then s [ ] else k ( )
|     changer (n, [ ]) p s k = k ( )
|     changer (n, x::R) p s k =
        if x <= n
        then
            changer (n-x, R)
             (fn A => p(x::A))
               (fn A => s(x::A))
                 (fn ( ) => changer (n, R) p s k)
        else
             changer (n, R) p s k
```

The proof that this function satisfies its specification? Again we can adapt the proof for `change`.

The value equations based on the above definition are:
For all types `t`, values `n`, `x` of type `int`, `L` and `R` of type `int list`, `p` of type `int list -> bool`, `s` of type `int list -> t` and `k` of type `unit -> t`:

(i) `changer (0, L) p s k = if p [ ] then SOME [ ] else k ( )`

(ii) `changer (n, [ ]) p s k = k ( )`
if `n > 0`

(iii)
```
changer (n, x::R) p s k =
   changer(n-x, R) (fn A => p(x::A))
        (fn A => s(x::A)) (fn ( ) => changer(n, R) p s k)
```
if `n` $> 0$ and `x` $\leq$ `n`

(iv) `changer (n, x::R) p s k = changer (n, R) p s k`
if `n` $> 0$ and `x` $>$ `n`.

Key fact (again): when `p` is total and `x` is an integer value, `fn A => p(x::A)` is a total function.

Is this really a more flexible function than the two previous ones?
Yes – we can easily derive the earlier functions from it:

```
fun change (n, L) p = changer (n, L) p (fn _ => true) (fn _ => false)

fun mkchange (n, L) p = changer (n, L) p SOME (fn ( ) => NONE)
```

These two (non-recursive!) function definitions satisfy the same specifications as the earlier recursive function definitions. Reason: this follows almost immediately from the specification of `changer`, which we proved (or at least claimed) is correct.

# 8 Summary

- We started with a function that returns a truth value, then a function that returns an (integer list) option, and finished with a function of a polymorphic type, that uses two "continuations", one to be called on "success" and one to be used in case of "failure". The polymorphic function can be "specialized" (by choosing an instance of its type, and picking appropriate success and failure continuations) to obtain the earlier functions as special cases.

- The use of success continuations, or failure continuations, or both, will turn out to be a very powerful methodology for solving a wide range of problems. This style is particularly potent for problems involving "backtracking", and many such problems are well known in AI settings or various combinatorial settings.

# 9 Self-test

1. Using equational reasoning, figure out the value of

   ```
   mkchange (10, [5,5]) (fn A => length A > 1).
   ```

   Check your answer using ML.

2. What do you expect to be the values of the following expressions? Confirm your expectations by equational reasoning and using ML.

   ```
   mkchange (10, [5,10,5]) (fn A => true)
   mkchange (10, [5,10,5]) (fn A => length A = 1)
   mkchange (10, [5,10,5]) (fn A => length A > 1)
   ```

3. Work out what happens when the following expressions are evaluated.

   ```
   changer (10, [10,5,5]) (fn A => length A > 1) SOME (fn ( ) => NONE)
   changer (10, [5,5,10]) (fn A => true) SOME (fn ( ) => NONE)
   changer (10, [10]) (fn A => true) (fn A => SOME(A@A)) (fn ( ) => NONE)
   changer (10, [5,5,10] (fn A => true) (fn A => 42::A) (fn ( ) => [42])
   ```

4. Consider what happens if we call `mkchange` with a list `L` that contains occurrences of 0. First, what is the value of

   ```
   mkchange (10, [0,0,5,0,0,0,10,0,5]) (fn A => true)
   ```
   ?

   Now answer the following questions:

   (i) Does the specification above tell us anything about this situation?

   (ii) How would you modify the specification to cover this situation?

   (iii) How would you adjust the correctness proof?