Today we’ll continue talking about uses of higher order functions, including:

1. Functions as data
2. Continuations, motivated by tail recursion

1 Functions as Data

Now that we know that functions can be returned as results from other functions, we can start to put this idea to use. Let’s consider the example of implementing lists as functions.

1.1 Dictionaries as Functions

Let’s say (hypothetically) that we’ve forgotten almost everything from the first two weeks of the course, except functions, but we still want to implement a data structure like dictionaries, which allow us to look up a value and get back a value associated with it.

Since all we have are functions, let’s represent a dictionary with keys of type `'a` and values of type `'b` as `'a -> 'b option` (after all, that reflects how we use a dictionary). The function returns SOME of the value if the key is in the dictionary, and NONE otherwise. Since we’ll need to compare the keys, we’ll say that the key type needs to be an equality type. (If we were doing something more reasonable like using a binary tree sorted on the keys, we’d actually need something stronger like a comparison function on the keys.)

```plaintext
type (''a, 'b) dict = ''a -> 'b option
```

The empty dictionary is fairly straightforward to implement:

```plaintext
val empty : (''a, 'b) dict = fn _ => NONE
```

How about lookup? Well, the representation of a function as a dictionary already corresponds perfectly to what lookup should do.
fun lookup (d : ('a, 'b) dict, k : 'a) : 'b option = d k

And, the last thing we need to be able to do is insert a new key-value pair into a dictionary. How should we do this? Well, we want to produce a function that returns the new value if the new key is given, and otherwise just falls back to being the old dictionary.

fun insert (d : ('a, 'b) dict, k : 'a, v : 'b) : ('a, 'b) dict =  
    fn k' => if k = k' then SOME v  
    else lookup (d, k)

2 Space Usage and Tail Recursion

Remember when we wrote two versions of reverse: reverse and its tail-recursive cousin fastRev. In analyzing the work of both, we determined fastRev to be faster because it doesn’t have to make all those calls to append. But being tail recursive is in itself a slight benefit (and can be a very large benefit in some languages).

So as to not get distracted by the asymptotic difference in work between reverse and fastRev, let’s look at an example that has the same work in its tail-recursive and non-tail-recursive forms.

2.1 Simple Sum

Recall sum:

val sum = foldr op+ 0

Expanding out the definition of foldr, we get the definition of sum from several lectures ago:

(* REQUIRES: true  
* ENSURES: sum l ==> the sum of the numbers in l  
*)
fun sum (l : int list) : int =  
    case l  
      of [] => 0  
      | x :: xs => x + sum xs

val 15 = sum [1,2,3,4,5]

If we do a little evaluation trace, we see that sum takes linear stack space:

    sum [1,2]  
    == 1 + sum [2]  
    == 1 + (2 + sum [])  
    == 1 + (2 + 0)  
    == 1 + 2  
    == 3
By “stack space”, we mean the part of the expression around the recursive call (e.g. $1 + (2 + \ldots)$). This is because a function call consumes stack space if and only if we have to do more work after the function returns, and if we make nested recursive calls, the stack space will be proportional to the number of recursive calls, which is proportional to the size of the expression. For a list of length $n$, the most additions we will have at any point is $n$. This is a problem for large $n$—most of the time, we don’t have a lot of stack space. If we run out, our program generally crashes in terrible ways. And if we want to add a long list, running out of space is a real possibility.

2.2 Tail Recursive Sum

Let’s write a tail recursive version `sumTC (tc here stands for tail call)` that uses constant space:

```sml
fun sum (L : int list) : int = 
  let
    fun sumTC (L : int list, a : int) : int =
      case L of
        [] => a
      | x :: xs => sumTC (xs , a + x)
  in
    sumTC (L , 0)
  end
val 15 = sum [1,2,3,4,5]
```

The function `sumTC` has an extra accumulator argument $a$, which stands for the sum so far. We assume inductively that it holds the partial result so far and ensure that assumption holds by updating it as we discover more partial results. Note that in the zero case we could have returned zero and still written type-correct code; that would be wrong because we’d be ignoring our inductive assumption about the accumulator $a$.

The notion of “the sum so far” is difficult to write down formally, so instead we write the spec more generally as $\text{sumTC}(L, s) = a + \text{sum}(L)$: regardless of what value we actually put in $a$, `sumTC` will add the sum of $L$ and return the result. Given that `sum` passes in $a = 0$, this implies that `sum` behaves correctly.

Here’s `sumTC` running on the same input from above:

```
sumTC ([1,2], 0)
== sumTC ([2], 1 + 0)
== sumTC ([2], 1)
== sumTC ([], 2 + 1)
== sumTC ([], 3)
== 3
```

`sumTC` is tail recursive: there is nothing left to do after the recursive call, to return from the outer call. Consequently, it uses constant space (whereas `sum` uses linear space\(^3\)—there is no

\(^3\)In SML/NJ, this is a lie. SML/NJ actually makes all functions tail recursive automatically, using a technique we are about to see.
memory necessary for storing “what’s left to do.” Note that the accumulator uses constant space only because values of type \texttt{int} are constant space—if the accumulator were a list (as in \texttt{fastRev}) then the accumulator would take up more space as time went on.

There is a common misconception that recursion is less space-efficient than loops, because of the stack space necessary to store the recursive calls. This example shows why this misconception is wrong:

\begin{quote}
\textit{recursion can be just as space-efficient as a loop!}
\end{quote}

Alternatively, we could just call our new favorite tail-recursive function, \texttt{foldl}:

\begin{verbatim}
val sum = foldl op+ 0
\end{verbatim}

We had said that the difference between \texttt{foldr} and \texttt{foldl} was that \texttt{foldr} folds from right to left and \texttt{foldl} folds from left to right. Since + is commutative, this doesn’t make a difference in the final answer, so we can just use the tail-recursive one.

3 Continuing

3.1 Question: Why do stacks even exist?

In most implementations of most programming languages, when we talk about stack space we’re talking about a small part of memory designated for storing function calls and their local variables, whereas most data are stored on the \textit{heap}, which is much, much larger. If running out of stack space is such a problem, the natural question is: Why make a distinction between stack and heap at all? We have space, we’re just not using it. The reason is having this distinction makes good sense for performance: memory is allocated and freed differently for the stack and heap, and storing things on the stack is faster when we can get away with it.

3.2 Question: Can we change that?

That being said, being able to use recursion as much as we want is a very powerful thing. And it’s better to focus on writing simple code first, then optimize for performance if necessary. So the next question is: How can we save stack space? for \texttt{sumTC} we managed to be clever and use an accumulator. We’re not always so clever, but in the general case we can always replace stack space with cheaper heap space using a technique called \textit{continuation-passing style}. We will explore multiple applications of continuations in the next few lectures, but the first application is automatically changing programs that use stack space into programs that use heap space. This can be done automatically by a compiler, and specifically is done by SML/NJ.

Let’s illustrate continuation-passing style with \texttt{sum}.

3.3 Example: Summing a List

Here’s the idea: we use an extra functional argument to represent \textit{what’s left to do}. This argument \(k\) is called a \textit{continuation}. It’s a plan for how to compute the overall result when given a partial result. To make this idea precise, we can write the spec as follows: \texttt{sum_cont}\texttt{’s} job is to compute the result, then pass it to \(k\). When we make a recursive call, we get to assume that our recursive call will do the same. Thus, by choosing which continuation we pass to our recursive call, we can do
extra work after the recursive call returns (i.e. when it calls the continuation). When programming
with continuations, it’s useful to first convince yourself why this is the right spec to have, then
focus on fulfilling that spec:

(* ENSURES: sum_cont L k == k (sum L) *)
fun sum_cont (L : int list) (k : int -> 'a) : 'a =
  case L of
    [] => k 0
  | x :: xs => sum_cont xs (fn a => k(x + a))

In the case for [], sum returns 0, so sum_cont feeds 0 to k. Note that the types prevent us from
making the mistake of simply returning 0: the only thing we can do to have sum return something
of 'a is to call k (though the types don’t prevent us from calling k with, say, 42; that’s up to us).

In the cons case, we make a recursive call on xs, writing down that, when you plug in the sum
of xs for s, you should add x to it, and feed the result to the continuation k. Right now, we don’t
worry about what k is going to do with the sum of x::xs, we just know it’s our job to pass that
sum to k.

Let’s do an evaluation trace, starting with the identity function fn x => x as the initial con-
tinuation:

\[
\begin{align*}
\text{sum_cont} & \ [1,2] \ (fn \ x \ => \ x) \\
& = \text{sum_cont} \ [2] \ (fn \ a \ => \ (fn \ x \ => \ x) \ (1 + a))
\end{align*}
\]

Let’s pause for a moment and make our lives easier. Notice we have the function (fn x => x)
applied to an argument 1 + s. This should simplify to 1 + s. Note this isn’t how the code actually
runs. In reality, this continuation doesn’t run until we reach the base case, but since we’re just
talking about equality, we can use referential transparency to simplify the code:

\[
\begin{align*}
& = \text{sum_cont} \ [2] \ (fn \ a \ => \ 1 + a) \\
& = \text{sum_cont} \ [] \ (fn \ a' \ => \ (fn \ s \ => \ (1 + a)) \ (2 + a')) \\
& = \text{sum_cont} \ [] \ (fn \ a' \ => \ 1 + (2 + a')) \\
& = \text{sum_cont} \ [] \ (fn \ a' \ => \ 3 + a') \\
& = (fn \ s' \ => \ 3 + s') \ 0 \\
& = 3 + 0 \\
& = 3
\end{align*}
\]

This makes the correspondence with the above trace for sum clear: in the second line of the
trace for sum, there is a 1 + ... . This is represented by the function (fn s => (1 + a)) in
the second step here. Similarly, in the next step there is 1 + 2 + ..., which corresponds to
(fn s' => (1 + (2 + a'))). That is, we have taken the stack (the part of the expression around
the recursive call), and represented it explicitly as a function!

\[\text{This is horrible name number three: first naming floating points “reals”; then naming things that don’t vary
“variables”; now naming the function that gets applied last the “initial” continuation. Note that the very last line
of the above evaluation trace finally fires off the initial continuation; on the first call, it gets buried deep in the
expression and stays there until the end.}\]
3.4 Continuation Passing Style

The function `sum_cont` above is an example of continuation passing style or CPS. CPS can mean different things, but for the purposes of this class, we will say that a function is in CPS if:

1. It takes at least one continuation as an argument (why would you want more than one? Wait and see...)

2. Any call that it performs to a CPS function is a tail call (i.e. is the last thing the function does, and the function doesn’t modify or inspect the return value of the function: this is what allows us to not allocate any new stack space for the function call.)

We will give a more detailed description of what is and isn’t CPS in lab and on the homework. If you’re unsure whether a function you wrote is in CPS, don’t hesitate to ask us.

3.5 Let’s continue folding

So far, what we’ve done seems rather less than pointless: we’ve taken our tail-recursive `sumTC` and made the code both more complicated and less efficient (it now takes linear heap space instead of constant heap and stack space).

In general though, being able to apply this transformation may get us some benefits. So, to convince you of how general this is, let’s rewrite a really general function in CPS. What’s the most general function we know that isn’t already tail-recursive? `foldr`! Let’s say we had some non-commutative function (like `::`) and we wanted to call `foldr` with it (so we couldn’t just use `foldl` which would give a different answer), but we wanted to do so in a tail recursive way.

In the base case, as with sum, we just want to pass the base case to the continuation. In the cons case, we want to call `foldr_cont` recursively and then apply `f` to the recursive result and `x`. The way we implement “and then” in CPS is by altering the continuation, so we do that by producing a new continuation that calls `f` “and then” does the rest of the stuff we already said we were going to “and then” (i.e. the current continuation `k`).

```haskell
fun foldr_cont (f : 'a * 'b -> 'b) (b : 'b) (l : 'a list) (k : 'b -> 'c) : 'c =
  case l of
  [] => k b
    | x :: xs => foldr_cont f b xs (fn b => k (f (x, b)))

fun foldr f b l = foldr_cont f b l (fn x => x)
```

For those keeping score at home, this recursive structure looks an awful lot like that of `foldl`, so just as when we saw that the structure of `sum` looked a lot like that of `foldr`, we rewrote `sum` in terms of `foldr`...

Let’s suggestively rearrange arguments to make it look even more like `foldl`...

```haskell
fun foldr_cont (f : 'a * 'b -> 'b) (k : 'b -> 'c) (l : 'a list) (b : 'b) : 'c =
  case l of
  [] => k b
    | x :: xs => foldr_cont f b xs (fn b => k (f (x, b)))

fun foldr f b l = foldr_cont f (fn x => x) l b
```
fun foldr_cont (f : 'a * 'b -> 'b) (k : 'b -> 'c) (l : 'a list) : 'b -> 'c =
    case l of
    [] => k
    | x :: xs => foldr_cont f b xs (fn b => k (f (x, b)))

fun foldr f b l = foldr_cont f (fn x => x) l b

fun foldr (f : 'a * 'b -> 'b) (b : 'b) (l : 'a list) =
    (List.foldl (fn (x, k) => (fn b => k (f (x, b)))) (fn x => x) l) b

Recall that in implementing foldl, the base case argument is essentially treated as an accumulator. In CPS functions, the accumulator is basically the continuation. So we call foldl with the initial accumulator being the identity function, and a function that transforms it into the new continuation. At the end, we’re left with a function that takes the base case and does all the work of foldr. We want to just apply that to b.

4 Continuations for Control

This still doesn’t actually buy us that much since SML/NJ already converts our functions to CPS automatically. But we can actually use continuations as a powerful control flow mechanism. Later, you’ll use this to implement a backtracking regular expression matcher.

4.1 Failure Continuations

Here, we consider a simpler example, namely a function that traverses a list and returns SOME of the shortest prefix of the list such that a predicate p is false on the next element. If p holds on all elements, the function returns NONE to indicate no such prefix exists.

First, a direct implementation.

(* findprefix : ('a -> bool) -> 'a list -> ('a list) option *)
fun findprefix p [] = NONE
    | findprefix p (x::xs) =
        if p x then (case findprefix p l of
                        SOME l => x::l
                        | NONE => NONE)
        else SOME []

Note an inefficiency in this function: if p is true for every element in the list, we case on the intermediate results of the recursive calls (all of which are NONE), passing along NONE all the way up to be returned as the overall result. Instead, we would like to return NONE immediately once we have reached the end of the list and found no element on which p is false.

Using continuation-passing style, we can rephrase a function that returns an option as a function that takes two continuations, a success continuation and a failure continuation, then calls the success continuation in the SOME case and the failure continuation in the NONE case.

Note that the function is now more efficient, as we don’t have to case on intermediate results. However, this is still a very simple example and may not actually buy us much. But we can actually use continuations as a powerful control flow mechanism. Later, you’ll use this to implement a backtracking regular expression matcher.

5. Continuations for Control

This still doesn’t actually buy us that much since SML/NJ already converts our functions to CPS automatically. But we can actually use continuations as a powerful control flow mechanism. Later, you’ll use this to implement a backtracking regular expression matcher.

4.1 Failure Continuations

Here, we consider a simpler example, namely a function that traverses a list and returns SOME of the shortest prefix of the list such that a predicate p is false on the next element. If p holds on all elements, the function returns NONE to indicate no such prefix exists.

First, a direct implementation.

(* findprefix : ('a -> bool) -> 'a list -> ('a list) option *)
fun findprefix p [] = NONE
    | findprefix p (x::xs) =
        if p x then (case findprefix p l of
                        SOME l => x::l
                        | NONE => NONE)
        else SOME []

Note an inefficiency in this function: if p is true for every element in the list, we case on the intermediate results of the recursive calls (all of which are NONE), passing along NONE all the way up to be returned as the overall result. Instead, we would like to return NONE immediately once we have reached the end of the list and found no element on which p is false.

Using continuation-passing style, we can rephrase a function that returns an option as a function that takes two continuations, a success continuation and a failure continuation, then calls the success continuation in the SOME case and the failure continuation in the NONE case.

Note that the function is now more efficient, as we don’t have to case on intermediate results. However, this is still a very simple example and may not actually buy us much. But we can actually use continuations as a powerful control flow mechanism. Later, you’ll use this to implement a backtracking regular expression matcher.

5. It buys us something since we can do a better job of converting our functions to CPS than SML/NJ can.
Think of it this way: when you write continuation-passing code, every time you make a recursive call, you have a choice: either you can “do something special” when the function returns (by passing a new anonymous function as the continuation) or you can “do whatever I was already going to do” by passing along the previous continuation \( k \). In contrast, if you use options, you always have to explicitly both the NONE and SOME case because all we can do with an option is case on it. Programming with continuations gives you the flexibility to tweak part of the control flow while leaving the rest of the control flow intact.

fun findpref_cont (p : 'a -> bool) (l : 'a list) (s : 'a list -> 'b) (k : unit -> 'b) : 'b = 
    case l of 
    | [] => k () 
    | x::xs => if p x then findpref_cont p xs (fn r => k (x::r)) k 
        else s [] 

fun findpref p l = findpref_cont p l SOME (fn () => NONE)

Note that in the “success” case (we find an element where \( p \) is false) we call the success continuation \( s \) on the expected initial answer (\( [] \) in this program). In the case of a “failure” (we do not find a place where \( p \) is false) we call the failure continuation \( k \) on... what is that, anyway?

### 4.2 Type unit

The code above uses a type called unit. What is unit? Let’s look at the values and operations:

**Values:** \( () \), often called the empty tuple or the unit tuple.

**Operations:** None! Nothing at all!

Why is such a type useful? It’s useful when we want to have a function, but we wish it didn’t have to take an argument (every function has to take an argument). By saying that a function takes an argument of type unit, we are documenting the fact that it doesn’t use the argument in any meaningful way.

### 4.3 General Use

It’s important to note that using continuation-passing style does not change the overall space usage of a program; it just moves space from stack to heap. Whether it’s necessary to do this depends on your implementation. For example, SML/NJ turns sum into sum_cont automatically, using what is called a *continuation-passing style transformation*. Thus, SML/NJ makes no distinction between stack space and heap space. So there is no point in writing sum_cont; it’s already being done for you!

On the other hand, if you choose the accumulator parameter cleverly (as in sumTC), you can sometimes reduce the overall space usage of your program. This is sometimes a useful technique if you’re running into real memory limitations, rather than artificial ones. It’s also useful because programs that use less space are often faster, too.

Continuations are a good trick to know, independently of this application to transforming stack space into heap space.
5 Correctness: Strengthening the IH

Let’s prove correctness of \texttt{sum\_cont}. Along the way, we’ll meet a new proof technique. Since we can transform any function into continuation-passing style, this proof is really a special case of the theorem showing that continuation-passing style transformation is valid.

5.1 First Attempt

\textbf{Theorem 1.} For all values \texttt{L:int list}, \texttt{sum\_cont L (fn x => x) \equiv sum L}.

This says that if we call \texttt{sum\_cont} with the identity function as the initial continuation, it behaves the same as above.

\textit{Proof.}

\textbf{Case for \texttt{[]}:} To show: \texttt{sum\_cont \texttt{[]} (fn x => x) \equiv sum \texttt{[]}.}

\textbf{Proof:}

\begin{verbatim}
    sum_cont [] (fn x => x)
    == case [] of [] => (fn x => x) 0
       | x::xs => sum_cont (fn s => (fn x => x)(x+s)) [step]
    == (fn x => x) 0 [step]
    == 0 [step]
    == case [] of [] => 0 | x::xs => x + sum xs [step,sym]
    == sum [] [step,sym]
\end{verbatim}

\textbf{Case for \texttt{x::xs}:} Let \texttt{x} be any \texttt{int} and \texttt{xs} be any \texttt{int list}.

\textbf{IH:} \texttt{sum\_cont xs (fn x => x) \equiv sum xs.}

\textbf{To show:} \texttt{sum\_cont (x::xs) (fn x => x) \equiv sum (x::xs).}

\textbf{Proof:}

\begin{verbatim}
    sum_cont (x::xs) (fn x => x)
    == sum_cont xs (fn s => (fn x' => x') (x + s)) [step]
    == sum_cont xs (fn s => (x + s)) [step, x+s valuable]
\end{verbatim}

At this point, we’d like to use the IH. We have an application of \texttt{sum\_cont} to \texttt{xs}, so in a sense we have no other hope. But we have a problem: the IH is stated only for the identity continuation, \texttt{fn x => x}. But we need an IH for the extended continuation \texttt{fn s => x + s}. So the proof breaks down.

\hfill \Box
5.1.1 What went wrong?

So what do we do? Is the code not correct?

To fix the proof, we need to strengthen the theorem statement (also called strengthening the IH). This way, the IH will give us more. It means we need to prove more in return, but that may be okay. Balancing between stronger IH’s and more difficult theorems is where a lot of the creativity in proofs comes in.

The above failed proof shows that we need to say something about an arbitrary continuation \( k \):

\[
\text{For all } k, \text{sum\_cont } l \ k \equiv ?
\]

But what can we say?

Follow the spec: \text{sum\_cont} behaves the same as computing \text{sum } l and then passing the result to \( k \) (but it does it in a different manner):

**Theorem 2.** For all values \( l:\text{int list}, k:\text{int->int} \), \text{sum\_cont } l \ k \equiv k(\text{sum } l).

5.1.2 Where the quantifier goes

As we saw above, the \( k \) changes in the recursive call. Thus, it’s not enough to fix a \( k \) at the outside and prove \text{sum\_cont } l \ k \equiv k(\text{sum } l) \) inductively. If we tried this, the the case for :: would be:

Case for \( \text{x::xs} \):

IH: \text{sum\_cont } xs \ k \equiv k(\text{sum } xs). To show: \text{sum\_cont } (\text{x::xs}) \ k \equiv k(\text{sum } (\text{x::xs})).

(IH is still not general enough.)

5.2 Correct Proof

The right way to do this is to quantify \( k \) in the predicate proved by induction: we prove “for all \( L, P(L) \)” where

\[
P(x) = \text{for all } k, \text{sum\_cont } x \ k \equiv k(\text{sum } x)
\]

Proof.

Case for \( [] \):

To show: for all \( k : \text{int -> int}, \text{sum\_cont } [] \ k \equiv k(\text{sum } []). \)

**Proof:** Let \( k \) be any value of type \( \text{int -> int}. \)

\[
\text{sum\_cont } [] \ k
\]

== case [] of [] => k 0 | x::xs => sum\_cont (fn s => k(x+s)) \ [step]

== k 0 \ [step]

== k(case [] of [] => 0 | x::xs => x+(sum xs)) \ [step,sym]

== k(sum []) \ [step,sym]
Case for $x::xs$:

**IH:** For all $k'$, $\text{sum\_cont\ } xs\ k' \cong k'(\text{sum}\ xs)$.

**To show:** For all $k$, $\text{sum\_cont\ } (x::xs)\ k \cong k(\text{sum}\ (x::xs))$.

**Proof:** Let $k$ be any value of type $\text{int} \rightarrow \text{int}$. 

$$\text{sum\_cont\ } (x::xs)\ k$$

$$= \text{sum\_cont\ } xs\ (\lambda s \Rightarrow k\ (x + s))\ \text{[step]}$$

Next we use the IH, *instantiating the quantifier by taking $k'$ to be $(\lambda s \Rightarrow k\ (x + s))$.*

$$= (\lambda s \Rightarrow k\ (x + s))\ (\text{sum}\ xs)\ \text{[IH, taking $k'$ as $(\lambda s \Rightarrow k\ (x + s))$]}$$

$$= k\ (x + (\text{sum}\ xs))\ \text{[step; sum is total; sum xs is valuable]}$$

$$= k\ (\text{sum}\ (x::xs))\ \text{[2 steps, symmetry]}$$

The important thing to notice is that the inductive hypothesis tells you more, but in return you have to prove more. This is called *strengthening the theorem statement* or *strengthening the IH*: you prove a stronger result than you want overall, to get the induction to go through. Our original statement that $\text{sum\_cont\ } l\ (\lambda x \Rightarrow x) \cong \text{sum}\ l$ follows as a corollary from the theorem easily.