15-150 Lecture 11:
Tail Recursion; Continuations

Lecture by Dan Licata

February 21, 2011

In this lecture we will discuss

- **space usage**: analyzing the memory it takes your program to run
- **tail calls and tail recursion**: when does a function run in constant stack space
- **Continuations**: using higher-order functions, you can explicitly represent "what to do next."

1 Space Usage and Tail Recursion

Recall `sum`:

```
fun sum (l : int list) : int =
   case l of
      [] => 0
    | x :: xs => x + sum xs
```

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.
For a list of length n, `sum` will have n additions on the stack at the high-water mark.
Let's write a *tail recursive* version `sum_tc` (`tc` here stands for *tail call*) that uses constant space:

```
(* Purpose: sum the list in constant space *)
local
    fun sumTC (l : int list, s : int) : int =
        case l of
            [] => s
```

```
          | x :: xs => sumTC (xs , s + x)
in
    fun sum (l : int list) : int = sumTC (l , 0)
end
val 15 = sum [1,2,3,4,5]
```

The function `sumTC` has an extra argument `s`, which stands for the sum so far. This is often refered to as an *accumulator parameter* (cf. `revTwoPiles`).

```
     sumTC ([1,2], 0)
|->* sumTC ([2], 1+0)
|->* sumTC ([2], 1)
|->* sumTC ([], 3 + 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)—there is no memory necessary for storing "what's left to do." Note that this depends on the fact that `int`'s are constant size; if they were infinite-precision integers, we'd still need the space for the accumulator parameter.

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:

recursion can be just as space-efficient as a loop!

Local. The function `sumTC` is an *auxiliary function*, in the sense that we don't anticipate anyone calling it besides `sum`. You can make the definition of `sumTC` *local* to `sum` by writing `local <decls> in <decls> en` This is like let, except the body is a sequence of declarations, not an expression. The function `sumTC` is visible in `sum` but not outside the `local` declaration.

2 Continuations

Traditional implementations of languages like C, Java, and C# distinguish between stack space and *heap* space (where all other data lives). Typically, the stack is much smaller than the heap. If a function recurs "too many" times, it will run out of stack space and crash, even though there is plenty of space available overall.

One solution to this problem is obvious: don't distinguish between stack and heap. Computers have plenty of memory, and often the memory needed to store "what's left to do" will fit just fine in the heap, even if it doesn't fit on the stack.

However, sometimes you want to execute deeply recursive code on a traditional implementation: some functional languages compile via C. Others, like Scala, compile to the JVM (Java Virtual Machine), or, like F#, to Microsoft's .NET common language runtime. In this situation, you have to deal with the stack/heap distinction, even though it's an accident of history.

We can do this using a general technique called *continuation-passing style*. Continuation-passing style has several applications (we'll see another one next time), but one use is to generically transform a program that uses stack space into a program that uses heap space. For `sum`, we were clever

and figured out how to do this using an accumulator parameter for the sum so far, but this took special knowledge about the function in question. Continuation-passing style is a generic way to achieve the same effect, without knowing about the code. Let's illustrate it with `sum`.

Here's the idea: we use an extra functional argument to represent *what's left to do*. This argument `k` is called a *continuation*.

```
fun sum_cont (l : int list) (k : int -> int) : int =
    case l of
       [] => k 0
     | x :: xs => sum_cont xs (fn s => k(x + s))
```

In the case for `[]`, `sum` returns `0`, so `sum_cont` feeds `0` to `k`. 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 (1) add `x` to it, and (2) feed the result to the continuation `k` (because, inductively, the caller is already something something that we're supposed to do with the sum of `x::xs`).

Let's do an evaluation trace, starting with the identity function `fn x => x` as the initial continuation:

```
    sum_cont [1,2] (fn x => x)
|->* sum_cont [2]   (fn s => (fn x => x) (1 + s))
|->* sum_cont []    (fn s' => (fn s => (fn x => x) (1 + s)) (2 + s'))
|->* (fn s' => (fn s => (fn x => x) (1 + s)) (2 + s')) 0
|->* (fn s => (fn x => x) (1 + s)) (2 + 0)
|->* (fn s => (fn x => x) (1 + s)) 2
|->* (fn x => x) (1 + 2)
|->* (fn x => x) 3
|->* 3
```

Because functions are values, we do not evaluate the continuation arguments until the end.

This makes the trace a little hard to read; if we use equivalence, it's easier. To do this, we need a new rule of equivalence:

> **Function extensionality:** For two functions `f g : A -> B`,
> `f` $\cong$ `g` if for all values `v:A`, `f v` $\cong$ `g v`.

This says that two functions are the same if they agree on all arguments. This is valid because (1) the only thing you can do with a function is apply it.

```
    sum_cont [1,2] (fn x => x)
≅   sum_cont [2] (fn s => (fn x => x) (1 + s))
≅   sum_cont [2] (fn s => (1 + s))
≅   sum_cont [] (fn s' => (fn s => (1 + s)) (2 + s'))
≅   sum_cont [] (fn s' => (1 + (2 + s')))
≅   (fn s' => (1 + (2 + s'))) 0
≅   (1 + (2 + 0))
≅   3
```

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 + s))`

in the second step here. Similarly, in the next step there is `1 + 2 + -`, which cprresponds to `(fn s' => (1 + (2 + s')))`. That is, we have taken the stack (the part of the expression around the recursive call), and represented it explicitly as a function!

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, SMLNJ turns `sum` into `sum_cont` automatically, using what is called a *continuating passing style transformation*. Thus, SMLNJ 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.

Continuations are a good trick to know, independently of this application to transforming stack space into heap space. Next class, we'll see that representing continuations explicitly can be useful for backtracking algorithms. Another application, which we won't get to in 150, is web programming: programs that interact over HTTP are inherently written in continuation-passing style, to account for the fact that the server program must terminate every time it passes control to the browser; letting a compiler transform a direct program into continuation-passing style makes web programming much more pleasant.

3 Correctness: Strengthening the IH

Let's prove correctness of `sum_cont`. Along the way, we'll meet a new proof technique.

Theorem 1. *For all* `l:int list`, `sum_cont l (fn x => x)` $\cong$ `sum l`.

This says that if we call `sum_cont` with the identity function as the initial continuation, it behaves the same as above.

Proof. **Case for []:**
To show: `sum_cont [] (fn x => x)` $\cong$ `sum []`.
Proof:

```
   sum_cont [] (fn x => x)
== case [] of [] => (fn x => x) 0 | ...     [step]
== (fn x => x) 0                            [step]
== 0                                        [step]
== sum []                                   [step^2]
```

Case for `x::xs`:
IH: `sum_cont xs (fn x => x)` $\cong$ `sum xs`.
To show: `sum_cont (x::xs) (fn x => x)` $\cong$ `sum (x::xs)`.
Proof:

```
   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]
```

At this point, we'd like to use the IH. But we have a problem: the IH is stated only for the identity continuation, `fn x => x`. But we need an IH for the extended continuation `fn s => x + s`. So the proof breaks down. □

To fix this, we need to *strengthen the theorem statement* (also called *strengthening the IH*). This way, the IH will give us more, but we need to prove more in return. Balancing this fulcrum is where a lot of the creativity in proofs comes in.

The above failed proof shows that we need to say something about a an arbitrary continuation `k`:

For all `k`, `sum_cont l k` ≅ ?

But what can we say?

The intuition is that `sum_cont` behaves the same as computing `sum l` and then passing the result to `k` (but it does it in a different manner):

Theorem 2. *For all values* `l:int list`, `k:int->int`,
`sum_cont l k` ≅ `k(sum l)`.

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 `sum_cont l k` ≅ `k(sum l)` inductively. If we tried this, the the case for `::` would be:

> **Case for `x::xs`:**
> IH: `sum_cont xs k` ≅ `k(sum xs)`. To show: `sum_cont (x::xs) k` ≅ `k(sum (x::xs))`.
>
> (IH is still not general enough.)

The right way to do this is to quantify `k` in the predicate proved by induction: we prove "for all `l`, P(`l`)" where

$$P(\texttt{x}) = \text{for all } \texttt{k}, \texttt{sum\_cont x k} \cong \texttt{k(sum x)}.$$

Proof. **Case for []:**
To show: for all `k`, `sum_cont [] k` ≅ `k(sum [])`.
Proof: Assume `k`.

```
   sum_cont [] k
== case [] of [] => k 0 | ...     [step]
== k 0                            [step]

   k(sum [])                      [step]
== k(case [] of [] => 0 | ...)    [step]
== k(0)                           [step]
```

Case for `x::xs`:
IH: For all `k'`, `sum_cont xs k'` ≅ `k'(sum xs)`.
To show: For all `k`, `sum_cont (x::xs) k` ≅ `k(sum (x::xs))`.

Proof: Assume `k`.

```
   sum_cont (x::xs) k
== sum_cont xs (fn s => k (x + s))   [step]
```

Next we use the IH, *instantiating the quantifier by taking* `k'` *to be* `(fn s => k (x + s))`.

```
== (fn s => k (x + s)) (sum xs)   [IH, taking k' = (fn s => k (x + s)) ]
== k (x + (sum xs))               [step; sum is total so sum xs is valuable]
== k (sum (x + xs))               [step^2]
```

□

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 `sum_cont l (fn x => x)` $\cong$ `sum l` follows as a corollary.