CSE341: Programming Languages

Lecture 6
Tail Recursion, Accumulators, Exceptions

Dan Grossman
Fall 2011

Two unrelated topics

1. Tail recursion
2. Exceptions

Recursion

Should now be comfortable with recursion:

- No harder than using a loop (whatever that is ☺)
- Often much easier than a loop
  - When processing a tree (e.g., evaluate an arithmetic expression)
  - Examples like appending two lists
  - Avoids mutation even for local variables
- Now:
  - How to reason about *efficiency* of recursion
  - The importance of *tail recursion*
  - Using an *accumulator* to achieve tail recursion
  - [No new language features here]

Call-stacks

While a program runs, there is a *call stack* of function calls that have started but not yet returned

- Calling a function `f` pushes an instance of `f` on the stack
- When a call to `f` to finishes, it is popped from the stack

These stack-frames store information like the value of local variables and "what is left to do" in the function

Due to recursion, multiple stack-frames may be calls to the same function

---

Example

```
fun fact n = if n=0 then 1 else n*fact(n-1)
val x = fact 3
```

fact 3	fact 3: 3*_	fact 3: 3*_	fact 3: 3*_
	fact 2	fact 2: 2*_	fact 2: 2*_
		fact 1	fact 1: 1*_
			fact 0

fact 3: 3*_	fact 3: 3*_	fact 3: 3*_	fact 3: 3*2
fact 2: 2*_	fact 2: 2*_	fact 2: 2*1	
fact 1: 1*_	fact 1: 1*1		
fact 0: 1			

Example Revised

```
fun fact n =
    let fun aux(n,acc) =
            if n=0
            then acc
            else aux(n-1,acc*n)
    in
       aux(n,1)
    end
val x = fact 3
```

Still recursive, more complicated, but the result of recursive calls *is* the result for the caller (no remaining multiplication)

The call-stacks

fact 3	fact 3:_	fact 3:_	fact 3:_
	aux(3,1)	aux(3,1):_	aux(3,1):_
		aux(2,3)	aux(2,3):_
			aux(1,6)

fact 3:_	fact 3:_	fact 3:_	fact 3:_
aux(3,1):_	aux(3,1):_	aux(3,1):_	aux(3,1):_
aux(2,3):_	aux(2,3):_	aux(2,3):_	aux(2,3):6
aux(1,6):_	aux(1,6):_	aux(1,6):6	
aux(0,6)	aux(0,6):6		

Etc…

An optimization

It is unnecessary to keep around a stack-frame just so it can get a callee's result and return it without any further evaluation

ML recognizes these *tail calls* in the compiler and treats them differently:

- Pop the caller *before* the call, allowing callee to *reuse* the same stack space
- (Along with other optimizations,) as efficient as a loop

(Reasonable to assume all functional-language implementations do tail-call optimization)

What really happens

```
fun fact n =
    let fun aux(n,acc) =
            if n=0
            then acc
            else aux(n-1,acc*n)
    in
       aux(n,1)
    end

val x = fact 3
```

fact 3	aux(3,1)	aux(2,3)	aux(1,6)	aux(0,6)

Moral

- Where reasonably elegant, feasible, and important, rewriting functions to be *tail-recursive* can be much more efficient
  - Tail-recursive: recursive calls are tail-calls
- There is also a methodology to guide this transformation:
  - Create a helper function that takes an *accumulator*
  - Old base case becomes initial accumulator
  - New base case becomes final accumulator

Another example

```
fun sum xs =
   case xs of
      [] => 0
    | x::xs' => x + sum xs'
```

```
fun sum xs =
    let fun aux(xs,acc) =
            case xs of
               [] => acc
             | x::xs' => aux(xs',x+acc)
    in
       aux(xs,0)
    end
```


And another

```
fun rev xs =
   case xs of
      [] => []
    | x::xs' => (rev xs) @ [x]
```

```
fun rev xs =
    let fun aux(xs,acc) =
            case xs of
               [] => acc
             | x::xs' => aux(xs',x::acc)
    in
       aux(xs,[])
    end
```

Actually much better

```
fun rev xs =
   case xs of
       [] => []
     | x::xs' => (rev xs) @ [x]
```

- For **fact** and **sum**, tail-recursion is faster but both ways linear time
- The non-tail recursive **rev** is quadratic because each recursive call uses append, which must traverse the first list
  - And 1+2+…+(length-1) is almost length*length/2 (cf. CSE332)
  - Moral: beware list-append, especially within outer recursion
- Cons is constant-time (and fast), so the accumulator version rocks

Always tail-recursive?

There are certainly cases where recursive functions cannot be evaluated in a constant amount of space

Most obvious examples are functions that process trees

In these cases, the natural recursive approach is the way to go
- You could get one recursive call to be a tail call, but rarely worth the complication

[See `max_constant` example for arithmetic expressions]

Precise definition

If the result of `f x` is the "immediate result" for the enclosing function body, then `f x` is a tail call

Can define this notion more precisely…
- A *tail call* is a function call in *tail position*
- If an expression is not in tail position, then no subexpressions are
- In `fun f p = e`, the body `e` is in tail position
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (but `e1` is not). (Similar for case-expressions)
- If `let b1 … bn in e end` is in tail position, then `e` is in tail position (but no binding expressions are)
- Function-call arguments are not in tail position
- …

Exceptions

An exception binding introduces a new kind of exception

```
exception MyFirstException
exception MySecondException of int * int
```

The **raise** primitive raises (a.k.a. throws) an exception

```
raise MyFirstException
raise MySecondException(7,9)
```

A handle expression can handle (a.k.a. catch) an exception
- If doesn't match, exception continues to propagate

```
SOME(f x) handle MyFirstException => NONE
SOME(f x) handle MySecondException(x,_) => SOME x
```


Actually…

Exceptions are a lot like datatype constructors…

- Declaring an exception makes a constructor for type **exn**
- Can pass values of **exn** anywhere (e.g., function arguments)
  - Not too common to do this but can be useful
- Handle can have multiple branches with patterns for type **exn**