CISS445 Lecture 13: OCAML Part 3

Yihsiang Liow

January 31, 2020

Table of contents I

Exceptions I

- Here's an example of creating exception and throwing exception:

```
exception SomethingBadHappened;;
let f = fun x -> match x with
  42 -> 0
| _ -> raise SomethingBadHappened
;;
print_int (f 42);;
print_int (f 41);;
```

- You catch an exception with try-with:

Exceptions II

```
let g x =
try
    (f x)
with
    SomethingBadHappened -> -1
;;
print_int (g 42);;
print_int (g 41);;
```

- OCAML has some predefined exception types (e.g. `Division_by_zero`)

Matching with guards I

- You can include a boolean condition with a pattern (guard).
- Here's a print function for tuple (n, d) representing a fraction:

```
let print_fraction x = match x with
  (_,  0) -> print_string "undefined"
| (0,  _) -> print_int 0
| (n,  1) -> print_int n
| (n, -1) -> print_int (-n)
| (n,  d) when d > 0 ->
             Printf.printf "%d/%d" n d
| (n,  d) when d < 0 ->
             Printf.printf "%d/%d" (-n) (-d)
;;
```

Matching with guards II

- OCAML thinks that I might have not covered all cases. Create exception and use it in a "catch all" case at the bottom:

```
exception IgnoreCase;;
let print_fraction x = match x with
  (_,  0) -> print_string "undefined"
| (0,  _) -> print_int 0
| (n,  1) -> print_int n
| (n, -1) -> print_int (-n)
| (n,  d) when d > 0 ->
             Printf.printf "%d/%d" n d
| (n,  d) when d < 0 ->
             Printf.printf "%d/%d" (-n) (-d)
| _       -> raise IgnoreCase
;;
```

Matching with guards III

- Exercise. Write the following functions for fractions represented by 2-tuples. Use pattern matching (with guards if necessary).

```
fraction_add f1 f2
fraction_subtract f1 f2
fraction_mult f1 f2
fraction_div f1 f2
fraction_eq f1 f2   true iff f1 is f2 (as fractions)
fraction_neq f1 f2  true iff f1 is not f2
fraction_gt f1 f2   true iff f1 > f2
fraction_ge f1 f2   true iff f1 >= f2
fraction_lt f1 f2   true iff f1 < f2
fraction_le f1 f2   true iff f1 <= f2
```

foldl and foldr I

- Suppose a,b,c,...,x,y,z is a list and # is some binary operator. I can form two possible expressions:

$$a \# (b \# (c \# (...(x \# (y \# z))...)))$$

or

$$(((...((a \# b) \# c) \# ...x) \# y) \# z)$$

- Very common. E.g., 1 + (2 + (3 + (4 + 5)))
- Sometimes the two above can be the same (depending on #)
- Example: The following are the same

$((1+2)+3)+4$	"folds to the left"
$1+(2+(3+4))$	"folds to the right"

foldl and foldr II

- Example: Are the following the same?

$((1-2)-3)-4$	"folds to the left"
$1-(2-(3-4))$	"folds to the right"

- Now to implement fold-right (foldr)

foldl and foldr III

- Recursively:

  `foldr f [a;b;c;d] = f a (foldr f [b;c;d])`

- What is the base case?

```
foldr f [a;b;c;d] = f a (foldr f [b;c;d])
                  = f a (f b (foldr f [c;d]))
                  = f a (f b (f c (foldr f [d])))
                  = f a (f b (f c (f d (foldr f []))))
```

  i.e., need to specify `foldr f []`. This will be a third parameter for `foldr`.

- Therefore here's an implementation of foldr:

foldl and foldr IV

```
let rec foldr f list base = match list with
    [] -> base
| x::xs -> f x (foldr f xs base)
;;
```

- The product and sum of integers:

```
let prodlist list =
    foldr (fun x -> (fun y -> x * y)) list 1
;;
let sumlist list =
    foldr (fun x -> (fun y -> x + y)) list 0
;;
```

foldl and foldr V

- What about `foldl`? Assignment ...
- Note: `foldr` is our earlier `reduce` function. Having both `foldl` and `foldr` allows us to decide the order of operation.
- In most languages with a `reduce` function, the implementation is fold to the right. In some languages, the base case for `reduce` is the first value of the input list (and not as a separate input value).

filter I

- The filter function accepts a boolean function and a list and return a new list of values satisfying the boolean function.
- For instance

```
filter (fun x -> x > 0) [1;3;-2;5;-1;6;-3;-1]
```

  returns `[1;3;5;6]`.
- Exercise. Implement `filter`.
- Exercise. Can the count function (i.e., `count xs x` returns the number of times `x` occurs in `xs`) be implemented with `foldr`, `map`, `filter`?
- Exercise. Can the `length` function (i.e. returns the numbers of values in a list) be implemented using `foldr`, `map`, `filter`?

filter II

- Exercise. Is it possible that `foldr`, `map`, `filter` are related? Can `map` be implemented using `foldr`? Can `filter` be implemented using `foldr`?

Tail recursion I

- Here's a recursive function to reverse a list:

```
let rec rev list = match list with
  [] -> []
| x::xs -> (rev xs)@[x]
;;
```

Tail recursion II

- Time complexity: Let $T(n)$ be the time taken to call `rev` with a list of length n.

$$T(n) = \begin{cases} T(n-1) + An + B & \text{if } n > 0 \\ C & \text{if } n = 0 \end{cases}$$

  where A, B, C are constants.

- Why?

Tail recursion III

- The $An + B$ is because of appending the $n - 1$ items after returning from recursion:

```
rev [a;b;c;...;z] = (rev [b;c;...;z]) @ [a]
                  = [z;...;c;b] @ [a]
```

  Now OCAML has to remove `b` from `[z;...;c;b]` and `cons` to `[a]`: But OCAML lists are singly linked – therefore removing `b` is $O(n)$. `cons` is $O(1)$. Therefore runtime is $T(n) = T(n-1) + An + B$.

- Hence

$$T(n) = O(n^2)$$

Tail recursion IV

- Now consider this version of reverse:

```
let rec rev2 list revlist = match list with
    [] -> revlist
  | x::xs -> rev2 xs (x::revlist)
;;
```

- Where does the idea come from? Do a trace ...

```
rev2 [1;2;3;4] [] = rev2  [2;3;4]  1::[]
                  = rev2  [3;4]    2::1::[]
                  = rev2  [4]      3::2::1::[]
                  = rev2  []       4::3::2::1::[]
                  = 4::3::2::1::[]
```

Tail recursion V

- Time complexity of `rev2`:

$$T(n) = \begin{cases} T(n-1) + A & \text{if } n > 0 \\ B & \text{if } n = 0 \end{cases}$$

  For `rev2`, the A in $T(n)$ is because of `cons` of 1 item.

- Hence $T(n) = O(n)$.

Tail recursion VI

- What's the difference?

```
let rec rev list = match list with
   [] -> []
| x::xs -> (rev xs)@[x]
;;
let rec rev2 list revlist = match list with
   [] -> revlist
| x::xs -> rev2 xs (x::revlist)
;;
```

- Algorithmically speaking `rev2` is better:
  - For `rev`: `xs @ ys` has a runtime of $O(n)$ where n is the length of list `xs`.
  - For `rev2`: `x::xs` has a runtime of $O(1)$.

Tail recursion VII

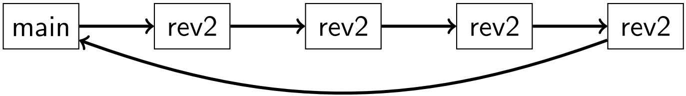
- Better to write this, hiding `rev2` another function say `rev`:

```
let rev list =
    let rec rev2 list revlist = match list with
      [] -> revlist
    | x::xs -> rev2 xs (x::revlist) in
    rev2 list []
;;
```

- Note that the `rev2` takes an extra parameter that builds the solution – the “difference list”.
- LISP programmers do this a lot.
- But not only is the algorithmic runtime of `rev2` better.
- There's something curious about tail recursion ...

Tail recursion VIII

- For `rev2`, the difference list at the last call is in fact the return value. Therefore there's really no need for function return until the last, i.e., the first `rev2` need not return, the second `rev2` need not return,... Only the last `rev2` needs to return.



- Why is this called tail recursion? The value to be computed is built up slowly and is finally obtained at the last function call (tail).
- So what?

Tail recursion IX

- This means that the second function call `rev2` can use the function frame of `rev1` in the stack segment, etc.
- Therefore a tail recursion will use less stack memory and not have stack overflow error. $O(n)$ stack memory is reduced to $O(1)$ if n is the number of recursive function calls.
- Plus you save on the time to allocate and deallocate frames in the stack segment.
- Certain languages are smart enough to figure this out and generate assembly code to take advantage of the above.
- Because of the simpler structure of a tail recursive function, some compiler can also convert it to a non-recursive function containing a loop.

Tail recursion X

- What about `rev`? The return value is used by `rev`:

```
let rec rev list = match list with
  [] -> []
| x::xs -> (rev xs)@[x]
;;
```

- So `rev` has to return to previous `rev` function call:

Tail recursion XI

- Make sure you see that `rev2` does not compute with the return value from a `rev` function call:

```
let rec rev2 list revlist = match list with
  [] -> revlist
| x::xs -> rev2 xs (x::revlist)
;;
```

Tail recursion XII

- Here's the `len` function with tail recursion:

```
let rec len2 list a = match list with
  [] -> a
| x::xs -> len2 xs (a + 1)
;;
```

- Now try to trace

```
len2 [1;2;3] 0;;
```

Tail recursion XIII

- Both `rev2` and `len2` carries an extra parameter (usually called **accumulator**).
- `rev2` and `len2` computes the return value progressively and keeps the value in the extra parameter.
- When the base case is reached, the accumulator will have the final result which is then returned.

Tail recursion XIV

- To make the length function easier to use, you can define it in terms of `len2`. You can hide the definition of `len2` in len using let-in

```
let len list =
    let rec len2 list a = match list with
      [] -> a
    | x::xs -> len2 xs (a + 1) in
    len2 list 0
;;
```

Tail recursion XV

- A tail recursion call does not require a return value for computation:

```
let rec f <blah> = match ...
  <base> -> <...>
| <recursive> -> f <smaller blah>
```

- If the recursive case looks like this:

```
<recursive> -> <...> + (f <smaller blah>)
```

or some operation is performed on `f <smaller blah>`, then it's not a tail recursion.

Tail recursion XVI

- Exercise. Write a sum function so that `(sum n)` computes $1 + \cdots + n$. Return 0 if $n \leq 0$. Use tail recursion. (Compare with non-tail.)
- Exercise. Write a `max` function so that `(max xs)` returns the maximum of value in `xs`. Use tail recursion. (Compare with non-tail.)
- Exercise. Write a `flip` function so that `(flip [1;2;3;4;5;6])` return `[2;1;4;3;6;5]`, i.e. consecutive pairs in the list are flipped. Use tail recursion. (Compare with non-tail.)

Tail recursion XVII

- Exercise. Write a `swap` function so that `(swap xs i)` will return the list `xs` except that the value at index `i` and index `i + 1` are swapped. For instance `(swap [1,2,3,4] 2)` returns `[1,2,4,3]`. Use tail recursion. (Compare with non-tail.)
- Exercise. Write a bubblesort function so that `(bubblesort xs)` returns `xs` sorted in the ascending order. Use tail recursion. (Compare with non-tail.)
- Exercise. Write an `insert` function so that `(insert xs i v)` returns `xs` except that `v` is inserted at index position `i`. For instance `(insert [1;3;5] 1 2)` returns `[1;2;3;5]` and `(insert [1;3;5] 3 2)` returns `[1;3;5;2]`. Use tail recursion. (Compare with non-tail.)

Structural recursion I

- A function is said to have **structural recursion** if the function is recursive and the recursive call depends on a recursion of the data structure of a parameter.
- Example:

```
let rec len list = match list with
    [] -> 0
  | x::xs -> 1 + (len xs)
  ;;
```

  In this example (`len x::xs`) depends on (`len xs`).
- So far the only recursive data structure we have used is the OCAML list. We will be talk about other data structures very soon.

Structural recursion II

- **Forward recursion** is recursion where recursion is called on substructures of a data structure (a parameter) and final result is built from the result of recursion on substructures. More or less the opposite of tail recursion.

Structural recursion III

- Exercise. Try these:

```
(+);;
(+) 3 5;;
```

  What should you try immediately? Rewrite the `sumlist` and `prodlist` functions from the previous notes. Also, rewrite them using tail recursion.

C++ I

- Recall the following recursive function:

$$\text{sum}(n) = \begin{cases} n + \text{sum}(n-1) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

- Here's the C++ version using tail recursion:

```
int sum1(const int n, const int acc=0)
{
    if (n == 0) return acc;
    else sum1(n - 1, acc + n);
}
```

C++ II

- I'm deliberately using `const int` so that you see the recursion does not require variables – just like OCAML recursive.
- Why write this C++ function in tail recursion?
- Because g++ can do tail recursion optimization if you use the -O2 optimization

  ```
  g++ main.cpp -O2
  ```

- Run `g++ main.cpp -O2 -S` on the following, read the assembly code, check that the compiler converted the function call into a loop.

C++ III

```
#include <iostream>
int sum1(const int n, const int acc)
{
    if (n == 0) return acc;
    else sum1(n - 1, acc + n);
}
int main()
{
    std::cout << sum1(5, 0) << '\n';
    return 0;
}
```

C++ IV

- Intel assembly with -O2:

```
        .file   "main.cpp"
        .text
        .p2align 4,,15
        .globl  _Z4sum1ii
        .type   _Z4sum1ii, @function
_Z4sum1ii:
.LFB1482:
        .cfi_startproc
        testl   %edi, %edi
        movl    %esi, %eax
        jle     .L6
        .p2align 4,,10
        .p2align 3
.L7:
        addl    %edi, %eax
        subl    $1, %edi
        jne     .L7
```

C++ V

```
.L6:
        rep ret
        .cfi_endproc
.LFE1482:
        .size   _Z4sum1ii, .-_Z4sum1ii
        .section .text.startup,"ax",@progbits
        .p2align 4,,15
        .globl  main
        .type   main, @function
main:
.LFB1483:
        .cfi_startproc
        subq    $24, %rsp
        .cfi_def_cfa_offset 32
        movl    $15, %esi
        movl    $_ZSt4cout, %edi
        call    _ZNSolsEi
        leaq    15(%rsp), %rsi
```

C++ VI

```
        movl    $1, %edx
        movq    %rax, %rdi
        movb    $10, 15(%rsp)
        call    _ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ost
        xorl    %eax, %eax
        addq    $24, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE1483:
        .size    main, .-main
        .p2align 4,,15
        .type    GLOBAL__sub_I__Z4sum1ii, @function
_GLOBAL__sub_I__Z4sum1ii:
.LFB1921:
        .cfi_startproc
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
```

C++ VII

```
        movl    $_ZStL8__ioinit, %edi
        call    _ZNSt8ios_base4InitC1Ev
        movl    $__dso_handle, %edx
        movl    $_ZStL8__ioinit, %esi
        movl    $_ZNSt8ios_base4InitD1Ev, %edi
        addq    $8, %rsp
        .cfi_def_cfa_offset 8
        jmp     __cxa_atexit
        .cfi_endproc
.LFE1921:
        .size   _GLOBAL__sub_I__Z4sum1ii, .-_GLOBAL__sub_I__Z4sum1ii
        .section .init_array,"aw"
        .align  8
        .quad   _GLOBAL__sub_I__Z4sum1ii
        .local  _ZStL8__ioinit
        .comm   _ZStL8__ioinit,1,1
        .hidden __dso_handle
```

C++ VIII

```
        .ident  "GCC: (GNU) 6.4.1 20170727 (Red Hat 6.4.1-1)"
        .section .note.GNU-stack,"",@progbits
```

C++ IX

- MIPS assembly with -O2:

```
        .file   1 ""
        .section .mdebug.abi32
        .previous
        .nan    legacy
        .module fp=32
        .module nooddspreg
        .abicalls
        .text
        .align  2
        .globl  sum1
        .set    nomips16
        .set    nomicromips
        .ent    sum1
        .type   sum1, @function
sum1:
        .frame  $sp,0,$31 # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000,0
```

C++ X

```
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro
        blez    $4,$L11
        move    $2,$5

$L8:
        addu    $2,$4,$2
        addiu   $4,$4,-1
        bne     $4,$0,$L8
        nop

$L11:
        j       $31
        nop

        .set    macro
        .set    reorder
```

C++ XI

```
        .end    sum1
        .size   sum1, .-sum1
        .align  2
        .globl  main
        .set    nomips16
        .set    nomicromips
        .ent    main
        .type   main, @function
main:
        .frame  $sp,0,$31 # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro
        j       $31
        move    $2,$0

        .set    macro
```

C++ XII

```
        .set    reorder
        .end    main
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.1) 5.4.0 20160609"
```

Tail recursion I

- Compare "usual" and tail recursion.

```
sum x::xs      = x + (sum xs)

sum2 x::xs a = sum2 xs (x + a)
```

```
len x::xs      = 1 + (len xs)

len2 x::xs a = len2 xs (1 + a)
```

```
inclist x::xs      = (x+1)::(inclist xs)

inclist2 x::xs a = inclist2 xs ((x+1)::a)
```