Compiler Construction: Supervision 1

Daniel Chatfield

January 30, 2015

Theoretical Exercise Set 1

1. Discuss how to eliminate recursion given a mutually tail-recursive functions. Transform the following by hand.

```
let rec is_even n =
    if n = 0
    then true
    else is_odd(n - 1)

and is_odd n =
    if n = 0
    then false
    else is_even(n - 1)
```

```
let is_even n =
    let rn = ref n
    in let _ = while !n > 2
      do
        rn := !rn - 2
      done
    in if !rn = 0
       then true
       else false
```

2. Again by hand, eliminate tail recursion from `fold_left`. Does your source-to-source transformation change the type of the function? If so, can you rewrite your code so that the type does not change?

```
(* fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)
let rec fold_left f accu l =
  match l with
      [] -> accu
  | a::l -> fold_left f (f accu a) l

(* sum up a list *)
let sum1 = fold_left (+) 0 [1;2;3;4;5;6;7;8;9;10]
```

3. Apply (by hand) the CPS transformation to the gcd code.

   Explain your results.

```
let rec gcd(m, n) =
    if m = n
    then m
    else if m < n
        then gcd(m, n - m)
        else gcd(m - n, n)

let gcd_test_1 = List.map gcd [(24, 638); (17, 289); (31, 1889)]
```

```
let rec gcd_cps(cnt, m, n) =
    if m = n
    then cnt m
    else if m < n
        then gcd_cps(cnt, m, n - m)
        else gcd_cps(cnt, m - n, n)
```

4. Environments are treated as functions in `interp_0.ml`.

   Can you transform these definitions, starting with defunctionalisation, and arrive at a list-based implementation of environments?

```
(* update : ('a -> 'b) * ('a * 'b) -> 'a -> 'b *)
let update(env, (x, v)) = fun y -> if x = y then v else env y

(* mupdate : ('a -> 'b) * ('a * 'b) list -> 'a -> 'b *)
let rec mupdate(env, bl) =
    match bl with
    | [] -> env
    | (x, v) :: rest -> mupdate(update(env, (x, v)), rest)

(* env_empty : string -> 'a *)
let env_empty = fun y -> failwith (y ^ " is not defined!\n")

(* env_init : (string * 'a) list -> string -> 'a *)
let env_init bl = mupdate(env_empty, bl)
```

```
type ('a, 'b) obear = EMPTY | INHABITED of 'a * 'b * ('a, 'b) obear

let apply bear y =
    match bear with
    | EMPTY -> failwith (y ^ " is not defined!\n")
    | INHABITED(x,v,rest) -> if x = y then v else apply rest y

let rec mupdate(env, bl) =
    match bl with
    | [] -> env
    | (x, v) :: rest -> mupdate(INHABITED(x,v,env), rest)

let env_init bl = mupdate(EMPTY, bl)




let update(env, new_item) = new_item :: env

let rec mupdate(env, bl) =
    match bl with
    | [] -> env
    | new_item :: rest -> mupdate(update(env, new_item), rest)

let rec env_lookup(env, y) =
    match env with
    | [] -> failwith (y ^ " is not defined!\n")
    | (x, v) :: rest -> if x = y then v else env_lookup rest y
```

5. Below is the code for (uncurried) map, with a test using fib. Can you apply the CPS transformation to map to produce map_cps? Will this map_cps still work with fib? If not, what to do?

```
(* map : ('a -> 'b) * 'a list -> 'b list *)
let rec map(f, l) =
    match l with
    | [] -> []
    | a :: rest -> (f a) :: (map(f, rest))

(* fib : int -> int *)
let rec fib m =
    if m = 0
    then 1
    else if m = 1
        then 1
        else fib(m - 1) + fib (m - 2)

let map_test_1 = map(fib, [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10])
```

```
let map(cnt, f, l) =
     match l with
     | [] -> cnt []
     | x::xs -> f (fun v -> map((fun v2 -> cnt (v :: v2)), f, (xs))) x
```

Practical Exercises

There are many possible extensions or modifications possible for the Slang.1 that be fun programming (nested comments, better verbose output, etc). However, I'll list only those exercises that I feel reinforce the main points of the lectures.

They are listed from easiest to hardest.

6. In this simple language, do we really need the return types of functions? Could we always infer them?

7. Add division to the arithmetic operations. How will you handle division by zero?

8. We have blurred the distinction between the defining language (OCaml) and the defined language (Slang.1) in several ways. For example, Slang.1 integers

are the ints supplied by OCaml on your machine. Now suppose that Slang.1 only allows 16-bit integers, where overflow or underflow should raise a run-time error. Can you modify the implementation to match this? Are there additional built-in functions that you might want to add to such a language?

9. Add mutual recursion to Slang1. Such as

   ```
   let g(x : int) : int = ... f(e) ...
   and f(z :int) : int  = .... g(e') ...
   in ... end
   ```

   This requires careful treatment of environments!

10. Consider adding references to Slang.1. We could do this in several ways, from simplest to most complex:

    - Allow references to be used only locally.
    - Allow refs to be passed to functions and then updated by the called function.
    - Allow refs to be returned by functions

    Can `interp_0.ml` be extended to handle each of these?