CSE216 Programming Abstractions

Continuation Passing Style

YoungMin Kwon

Continuation

- Continuation
  - A single parameter function that represents the rest of the program
  - A function written in CPS takes an extra parameter: continuation
  - The function passes its result to the continuation

Sum

- Sum: recursive function

```
let rec sum n =
    if n = 0
    then n
    else n + sum (n-1)
```

- Sum: Continuation Passing Style

```
let sum n =
    let rec iter n k =
```

Sum

- Sum: recursive function

```
let rec sum n =
    if n = 0
    then n
    else n + sum (n-1)
```

- Sum: Continuation Passing Style

```
let sum n =
    let rec iter n k =
        if n = 0
        then k n
        else iter (n-1) (fun x -> k (x + n)) in
    iter n (fun x -> x)
```

Fibonacci Number

- fib is a recursive function that computes the n-th Fibonacci number

```
let rec fib n =
    if n <= 1
    then n
    else fib (n-1) + fib (n-2)
```

- Rewrite fib in CPS

```
let fib n =
    let rec iter n k =
```

Symbolic Derivative

- Symbolic Derivative
  - Given a symbolic expression, return its symbolic derivative

- Differentiation

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

Symbolic Derivative

```
type expr = N of int              (*number*)
          | X | Y | Z             (*variable*)
          | Add of expr * expr    (*add expr*)
          | Mul of expr * expr    (*mul expr*)

let add a b = (*add two exprs*)
    match (a, b) with
    | (N 0, y) -> y
    | (x, N 0) -> x
    | (N x, N y) -> N (x + y)
    | _ -> Add (a, b)

let mul a b = (*multiply two exprs*)
    match (a, b) with
    | (N 0, y) -> N 0
    | (N 1, y) -> y
    | (x, N 0) -> N 0
    | (x, N 1) -> x
    | (N x, N y) -> N (x * y)
    | _ -> Mul (a, b)
```

Symbolic Derivative

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

```
(*derivative of expr w.r.t. var*)
let rec deriv expr var =
    match expr with
    | N x -> N 0
    | X | Y | Z  -> if expr = var then N 1 else N 0
    | Add (a, b) -> add (deriv a var) (deriv b var)
    | Mul (a, b) -> add (mul a (deriv b var)) (mul (deriv a var) b)
```

```
(*TODO: rewrite deriv in CPS*)
let rec deriv expr var k =
```

Symbolic Derivative

```
(*derivative of expr w.r.t. var*)
let rec deriv expr var =
    match expr with
    | N x -> N 0
    | X | Y | Z  -> if expr = var then N 1 else N 0
    | Add (a, b) -> add (deriv a var) (deriv b var)
    | Mul (a, b) -> add (mul a (deriv b var)) (mul (deriv a var) b)

(*TODO: rewrite deriv in CPS*)
let rec deriv expr var k =
    match expr with
    | N x -> k (N 0)
    | X | Y | Z  -> if expr = var then k (N 1) else k (N 0)
    | Add (a, b) -> deriv a var (fun da ->
                    deriv b var (fun db ->
                    k (add da db)))
    | Mul (a, b) -> deriv a var (fun da ->
                    deriv b var (fun db ->
                    k (add (mul a db) (mul da b))))
```

Symbolic Derivative

```
(*convert expr to string*)
let rec to_str expr =
    let open Printf in
    match expr with
    | N a -> sprintf "%d" a
    | X -> "x"    | Y -> "y"    | Z -> "z"
    | Add (a, b) -> sprintf "(%s + %s)" (to_str a) (to_str b)
    | Mul (a, b) -> sprintf "%s * %s"   (to_str a) (to_str b)


(*TODO: rewrite to_str in CPS*)
let rec to_str expr k =
```

Symbolic Derivative

```
(*convert expr to string*)
let rec to_str expr =
    let open Printf in
    match expr with
    | N a -> sprintf "%d" a
    | X -> "x"    | Y -> "y"    | Z -> "z"
    | Add (a, b) -> sprintf "(%s + %s)" (to_str a) (to_str b)
    | Mul (a, b) -> sprintf "%s * %s"   (to_str a) (to_str b)

(*TODO: rewrite to_str in CPS*)
let rec to_str expr k =
    let open Printf in
    match expr with
    | N a -> k (sprintf "%d" a)
    | X -> k "x"    | Y -> k "y"    | Z -> k "z"
    | Add (a, b) -> to_str a (fun sa ->
                    to_str b (fun sb ->
                    k (sprintf "(%s + %s)" sa sb)))
    | Mul (a, b) -> to_str a (fun sa ->
                    to_str b (fun sb ->
                    k (sprintf "%s * %s" sa sb)))
```