CMSC 330: Organization of Programming Languages

Lambda Calculus and Types

Introduction

- We've seen that several language conveniences aren't strictly necessary
  - Multi-argument functions (use currying or tuples)

```
let fst (x,y) = x

let fst p =
  match p with
  (x,y) -> x
```

```
let fst x y = x

let fst x =
  fun y -> x
```


Introduction

- Loops (use recursion)
- Side-effects

```
r = 0;
for (i = 0; i < n; i++) {
  r += i;
}
```

```
let rec sum i r =
  if i >= n then r
  else sum (i+1) (r+i)
in
sum 0 0
```


Introduction

Goal: Come up with a "core" language that's as small as possible and still Turing complete

This will give a way of illustrating important language features and algorithms

Lambda Calculus Syntax

- A lambda calculus expression is defined as

e ::= x	variable
| λx.e	function
| e e	function application

- λx.e is like `(fun x -> e)` in OCaml
- That's it! All there is is higher-order functions

Conventions

- The scope of λ extends as far to the right as possible

  λx. λy.x y is λx.(λy.(x y))

- Function application is left-associative

  x y z is (x y) z

  – Same rule as OCaml

Operational Semantics

An interpreter in "math"

- All we've got are functions, so all we can do is call them
- To evaluate (λx.e1) e2
  - Evaluate e1 with x bound to e2

beta-reduction

$$(\lambda x.e1)\ e2 \rightarrow e1[x/e2]$$

- e1[x/e2] is e1 where occurrences of x are replaced by e2
- Slightly different than the environments we saw for Ocaml
  - apply a substitution instead of carry an environment
- We allow reductions to occur anywhere in a term

Examples

- (λx.x) z → z
- (λx.y) z → y
- (λx.x y) z → zy
  - A function that applies its argument to y
- (λx.x y) (λz.z) → (λz.z) y → y
- (λx.λy.x y) z → λy.z y
  - A curried function of two arguments that applies its first argument to its second
- (λx.λy.x y) (λz.zz) x →λy.((λz.zz)y)x → (λz.zz)x → xx

Syntactic sugar for local declarations

let x = e1 in e2

can be written as

(λx.e2) e1

Static Scoping and Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - The rightmost "x" refers to the second binding
  - This is a function that takes its argument and applies it to the identity function
- This function is "the same" as (λx.x (λy.y))
  - Renaming bound variables consistently is allowed

alpha-conversion (alpha-renaming)

Ex. λx.x = λy.y = λz.z λy.λx.y = λz.λx.z

Static Scoping (cont'd)

- How about the following?
  - (λx.λy.x y) y → ?
  - When we replace y inside, we don't want it to be "captured" by the inner binding of y
- This function is "the same" as (λx.λz.x z)

Beta-Reduction, Again

- Whenever we do a step of beta reduction...
  - (λx.e1) e2 → e1[x/e2]
  - ...alpha-convert variables as necessary
- Examples:
  - (λx.x (λx.x)) z = (λx.x (λy.y)) z → z (λy.y)
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z