Lecture 8

Lexical Scope and Function Closures and Interpreters

Lexical (Static) Scope vs Dynamic Scope

- Lexical scope : use environment where function was defined
- Dynamic scope : use environment where function is called
- In the early days of PL, folks pondered: *Which rule is better?*
    - Experience has shown that lexical scope is *almost always* the right default

Passing a Function

```
(* 1   *) fun f g =
(* 1 a *)   let val x = 3 in
(* 1 b *)   g 2 end
(* 2   *) val x = 4
(* 3   *) fun h y = x + y
(* 4   *) val z = f h
```

- "Trust the rule": Line 3 binds **h** to a closure:
  - Code: take **y** and have body `x + y`
  - Environment: `x ↦ 4, f ↦ <closure>,` … (anything else in scope)
  - So this closure will always add **4** to its argument
- So Line 4 binds `z` to **6**
  - Note Line 1a can't affect anything! Can/should safely delete it.

Currying and Partial Application

- Recall function types: `t1 -> t2 -> … -> tN`
    - Functions of this type "take `N` arguments of types `t1`, `t2`, …, `tN`"
- More precisely though: every SML function takes only 1 argument !!
    - Technically "take a `t1` and return a function that takes a `t2` and return a function that takes …"
- Means we can apply a function to "just some of its arguments"

Why Lexical Scope?

- Lexical scope : use environment where function was defined
- Dynamic scope : use environment where function is called
- In the early days of PL, folks pondered: *Which rule is better?*
  - Experience has shown that lexical scope is *almost always* the right default
- Let's consider 3 precise, technical reasons why
  - Not a matter of opinion!

Why Lexical Scope?

1. Functions meaning does not depend on variable names, only “shape”

```
fun f y =
Let val x = y + 1 in
  fn q => x + y + q end
```

```
fun f g =
  let val x = 3 in
    g 2 end
```

Example: can change **f** to use **w** instead of **x**

- Lexical scope : cannot change behavior of function
- Dynamic scope : depends on the environment of the caller

Example: can remove unused variables

- Lexical scope : variable unused, cannot change behavior
- Dynamic scope : some g may use x and depend on it being 3
    - WEIRD

Recomputation

These both work and are equivalent

```
fun all_shorter (xs,s) =
  filter(fn x => String.size x < String.size s) xs

fun all_shorter' xs s =
   let val n = String.size s in
   filter(fun x -> String.length x < n) xs
```

First version computes `String.sizes` repeatedly (once per x in `xs`)

Second version computes `String.size  s` once before filtering

- No new features! Just new use of closures

Why Lexical Scope?

Functions can be type checked and reasoned about where they are defined

- Example: dynamic scope tries to add string and has unbound variable **y**

```
val x = 1
fun f y =
   let val x = y + 1 in
   fn q => x + y + q end
val x = "hi"
val g = f 4
val z = g 6
```

Does Dynamic Scope Even Exist?

- Lexical scope is definitely the right default, seen in most languages
- Dynamic scope is occasionally convenient in some situations
  - Allows code to “just bind variables used in another function” to change behavior without passing parameters
    - Can be convenient, but also a nightmare
  - Some languages (including Racket!) have special features (dynamic variables in Scala)
- If you squint, exception handling is similar to dynamic scope
  - `raise e` jumps to “most recent” (dynamically registered) handler
  - Does not need to be syntactically inside the handler!

Interpreters

Addition

Syntax: `e1 + e2`

- Where `e1` and `e2` are expressions

Type Checking

- If has `e1` has type `int` and `e2` has type `int`
- Then `e1 + e2` has type `int`
- Else, report error and fail

Evaluation

- If `e1` evaluates to value `v1` and `e2` evaluates
- Then `e1 + e2` evaluates to the sum of `v1` and `v2`

But what if `e1` or `e2` do not evaluate to `int`s?

Type checking ensures they will be `int`s!

Comparison (less than)

Syntax: `e1 < e2`

- Where `e1` and `e2` are expressions

Type Checking

- If has `e1` has type `int` and `e2` has type `int`
- Then `e1 < e2` has type `bool`
- Else, report error and fail

Evaluation

- If `e1` evaluates to value `v1` and `e2` evaluates to value `v2`
- Then `e1 < e2` evaluates to **`true`** if `v1` is less than `v2`, and **`false`** otherwise

Conditionals (if-then-else)

Syntax: `if e1 then e2 else e3`

- Where `e1` and `e2` are expressions

Type Checking

- If has `e1` has type `bool` and `e2` and `e3` have the same type `t`
- Then `if e1 then e2 else e3` has type `t`
- Else, report error and fail

Evaluation

- If `e1` evaluates to **`true`**, then return result of evaluating `e2`
- If `e1` evaluates to **`false`**, then return result of evaluating `e3`

Pairs (2-tuples) : Build

Syntax: `(e1, e2)` where `e1` and `e2` are expressions

Type Checking

- If has `e1` has type `t1` and `e2` has type `t2`
- Then `(e1, e2)` has type `t1 * t2`
- Else, report error and fail (happens only if e1 or e2 does not type-check)

Evaluation

- If `e1` evaluates to value `v1` and `e2` evaluates to value `v2`
- Then `(e1, e2)` evaluates to `(v1, v2)`

Pairs (2-tuples) : Use

Syntax: `#1 e` and `#2 e`

- Where `e` is an expression

Type Checking

- If has `e` has type `t1 * t2`, then
- `#1 e` has type `t1`
- `#1 e` has type `t2`

Evaluation

- If `e` evaluates to a pair of values `(v1, v2)`, then
- `#1 e` evaluates to `v1`
- `#2 e` evaluates to `v2`

Function Bindings: Evaluation Rules

```
fun f ((x1 : t1),…,(xN : tN)) = e
```

Evaluation rules:

- *Nothing to do! Function are values!*
    - Real story a bit more nuanced, but we will get to it
- Add `f` to the dynamic environment so later expressions can call
    - `f` is bound to the function being defined here
- And for recursion, then `f` also bound within `e`

Function Calls

`e0 (e1,…, eN)`

Evaluation

1. Evaluate `e0` in current dynamic environment
   - Since `e0` type checked, the result will be a function
   - Let's call the result `fun f ((x1:t1),…,(xN:tN)) = e`
2. In current dynamic environment, evaluate

   `e1` to value `v1`, … , `eN` to value `vN`
3. The `e0 (e1,…, eN)` call now evaluates to the result of evaluating `e`

   ***In an extended dynamic environment with* `x1` ↦ `v1, …, xN` ↦ `vN`**