closure simulations

Daniel Jackson

For now, all rights reserved. Daniel Jackson, 2012
evaluating functions
two phases

creation
› function expression evaluated

application
› function body evaluated

evaluation order for applications
› first evaluate arguments, left to right
› then evaluate body
evaluating the body

what environment is body evaluated in?
› same environment application is evaluated in?

let’s see!
› hmm...

```javascript
> x = 1
1
> f = (function (x) {return function () {return x;};}) (x)
function () {return x;}
> f()
1
> x = 2
2
> f()
1
```
two environments

when function is created
› keeps environment as a property
› called ‘function scope’
› uses this environment to evaluate body in

what about arguments?
› new environment (‘frame’) with bindings for args
› linked to function scope
aah, nostalgia!

Figure 3.11
Sqrt procedure with internal definitions.

Expression (sqrt 2) where the internal procedure good-enough? has been called for the first time with guess equal to 1.

Observe the structure of the environment. Sqrt is a symbol in the
examples
example 1

> f = function () {return x;}
function () {return x;}
> x = 1
1
> f()
1
> x = 2
2
> f()
2

what happens here?
› function scope is top-level environment
› assignment to x modifies binding in top-level environment
› so in this case x refers to x of application environment too
simulating example 1

```javascript
> f = function () {return x;}
function () {return x;}
> x = 1
1
> f()
1
> x = 2
2
> f()
2
```
what happens here?

- function scope is top-level environment
- when application is evaluated, argument x is bound to 2
- local x said to shadow global x
simulating example 2

```
> f = function (x) {return x;}
function (x) {return x;}
> x = 1
1
> y = 2
2
> f(y)
2
```
what happens here?
› when f is applied, x is bound to 1 in new frame
› anonymous function has scope with x bound to 1
› assignment to top-level x does not modify this scope
simulating example 3

1. x = 1 (Activation) (Env) (Binding) bindings
   - context
   - var
   - value
   - var
   - value

2. f = ... (Activation) (Function)
   - _scope
   - _proto
   - var
   - value
   - var
   - value

2.1 return function ... ( Activation) (Function)
   - _scope
   - _proto
   - var
   - value
   - var
   - value

3. x = 2 (Activation) (Function)
   - body

4. f() (Activation) (Function)
   - return x

4.1 return x (Activation) (Function) (Env) (Binding) bindings
   - context
   - var
   - value
   - var
   - value

> x = 1
1
> f = (function (x) {
   return function () {
     return x;
   };
}) (x)
function () {return x;}
> x = 2
2
> f()
1
what if we modify x?

› when f is applied, x is bound to 0 in new frame
› anonymous function has scope with x bound to 0
› this ‘internal’ x is updated every time f is called
f = (function (x) {
    return function () {
        x += 1; return x;
    }
})(0)

1. f = ...
   (Activation)
   (Env)
   (Function)
   creates _scope
   body

2.1 return function ...
   (Activation)
   (Function)
   creates _proto
   bindings
   var
   value

3. f()
   (Activation)
   (Env)
   (Function)
   (Binding)
   creates _scope

> f = (function (x) {
    return function () {
        x += 1; return x;
    }
})(0)
function (){
    x += 1; return x;
}
> f()
1
lexical vs dynamic scoping
a language design question

what does this print?

- **lexical** scoping: 1, 2
- **dynamic** scoping: 3, 1

lexical scoping now preferred

- harder to implement
- better for programmer