Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 16

- Shallow and deep binding
- Returning subroutines
- Object Closures

“Referencing” (“Non-local”) Environments

- If a subroutine is passed as an argument to another subroutine, when are the static/dynamic scoping rules applied?
    1) When the reference to the subroutine is first created (i.e. when it is passed as an argument)
    2) Or when the argument subroutine is finally called
- That is, what is the *referencing environment* of a subroutine passed as an argument?
    – Eventually the subroutine passed as an argument is called and may access non-local variables which by definition are in the referencing environment of usable bindings
- The choice is fundamental in languages with dynamic scope: **deep binding (1)** vs **shallow binding (2)**
- The choice is limited in languages with static scope

Effect of Deep Binding in Dynamically-Scoped Languages

- The following program demonstrates the difference between deep and shallow binding:

```
function older(p:person):boolean
  return p.age > bound
procedure show(p:person,c:function)
  bound:integer
  bound := 20
  if c(p)
    write(p)
procedure main(p)
  bound:integer
  bound := 35
  show(p,older)
```

Program execution:

```
main(p)
  bound:integer
  bound := 35
  show(p,older)
    bound:integer
    bound := 20
    older(p)
      return p.age>bound
    if return value is true
      write(p)
```

Deep binding

Program prints persons older than 35

Effect of Shallow Binding in Dynamically-Scoped Languages

- The following program demonstrates the difference between deep and shallow binding:

```
function older(p:person):boolean
  return p.age > bound
procedure show(p:person,c:function)
  bound:integer
  bound := 20
  if c(p)
    write(p)
procedure main(p)
  bound:integer
  bound := 35
  show(p,older)
```

Program execution:

```
main(p)
  bound:integer
  bound := 35
  show(p,older)
    bound:integer
    bound := 20
    older(p)
      return p.age>bound
    if return value is true
      write(p)
```

Shallow binding

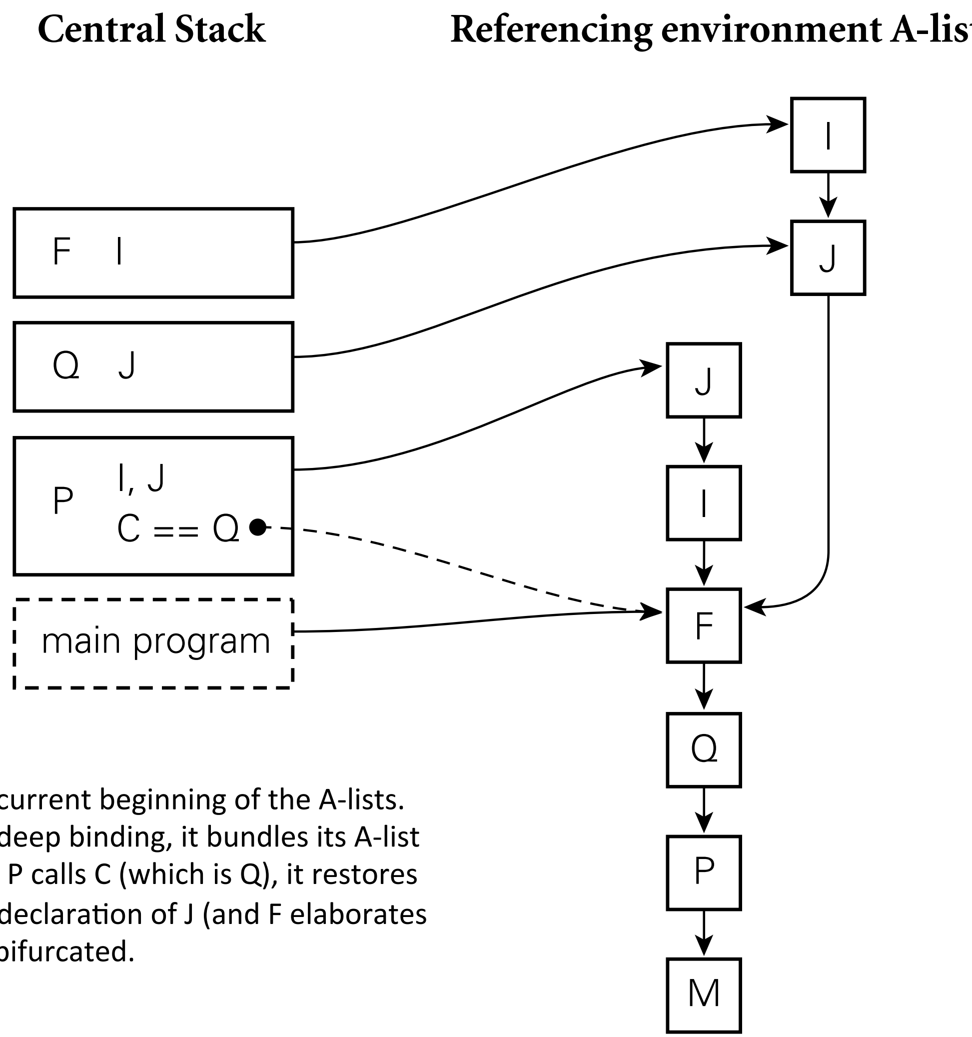
Program prints persons older than 20

Implementing Deep Bindings with Subroutine Closures

- Implementation of *shallow binding* obvious: look for the last activated binding for the name in the stack
- For *deep binding,* the referencing environment is bundled with the subroutine as a *closure* and passed as an argument
- A subroutine closure contains
  - A pointer to the subroutine code
  - The current set of name-to-object bindings
- Possible implementations:
  - With Central Reference Tables, the whole current set of bindings may have to be copied
  - With A-lists, the head of the list is copied

Closures in Dynamic Scoping implemented with A-lists

```
procedure P(procedure C)
        declare I, J
        call C

procedure F
        declare I

procedure Q
        declare J
        call F

-- main program
        call P(Q)
```



Each frame in the stack has a pointer to the current beginning of the A-lists. When the main program passes Q to P with deep binding, it bundles its A-list pointer in Q's closure (dashed arrow). When P calls C (which is Q), it restores the bundled pointer. When Q elaborates its declaration of J (and F elaborates its declaration of I), the A-list is temporarily bifurcated.

Denotational semantics for **deep/shallow binding** with **dynamic scoping** (1)

Syntax

Procedures have at most one parameter, which is a procedure name

Decl ::= … | **proc** Ide {Com} | **proc** Ide (Ide) {Com} *// Declaration*

Com ::= … | {Decl; Com} | **call** Ide | **call** Ide (Ide) *// Block, invocation*

Semantic domains

Procedures without parameters

Proc0 = **Env** → Store → Store

Procedures with one proc parameter

Proc1 = Proc0 → **Env** → Store → Store

Dval = … + Proc0 + Proc1…

Semantic interpretation functions

D: Decl → Env → Store → (Env x Store)

C: Cmd → Env → Store → Store

Semantics: no parameter

D{**proc** *p*{c}} r s = (r[*C*{*c*} /p], s)

C{**call** *p*} r = (r(p) as **Proc0**) **r**

Denotational semantics for **deep/shallow binding** with **dynamic scoping** (2)

Syntax

Procedures have at most one parameter, which is a procedure name

Decl ::= … | **proc** Ide {Com} | **proc** Ide (Ide) {Com} *// Declaration*

Com ::= … | {Decl; Com} | **call** Ide | **call** Ide (Ide) *// Block, invocation*

Semantic domains

Procedures without parameters

Proc0 = **Env** → Store → Store

Procedures with one proc parame

Proc1 = Proc0 → **Env** →Store → S

Dval = … + Proc0 + Proc1…

Semantic interpretation function

D: Decl → Env → Store → (Env x S

C: Cmd → Env → Store → Store

Semantics: one procedural parameter, dynamic scoping

D{**proc** *p*(*q*){c}} r s = (r[k /p], s)

where k = *λd:Proc0. λr'.C*{*c*} r'[d/q]

Shallow binding

C{**call** *p*(h)} = (r(p) as Proc1) (r{h} as Proc0)

Deep binding

C{**call** *p*(h)}r =

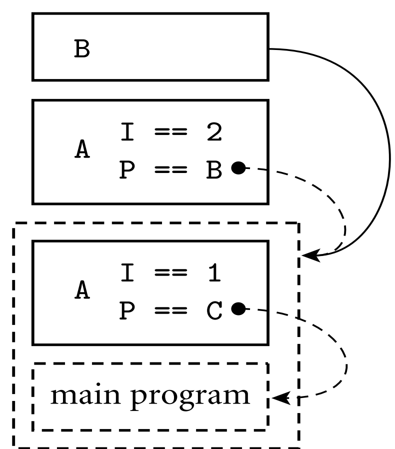
(r(p) as Proc1) (*λr'.(*r{h} as Proc0)r)r

Deep/Shallow binding with **static** scoping

- Not obvious that it makes a difference. Recall:
- **Deep binding**: the scoping rule is applied when the subroutine is passed as an argument
- **Shallow binding**: the scoping rule is applied when the argument subroutine is called
- In both cases non-local references are resolved looking at the static structure of the program, so refer to the same binding declaration
- **But in a recursive function the same declaration can be executed several times: the two binding policies may produce different results**
- No language uses shallow binding with static scope
- Implementation of deep binding easy: just keep the static pointer of the subroutine in the moment it is passed as parameter, and use it when it is called

Deep binding with **static scoping**: an example in Pascal

```
program binding_example(input, output);

procedure A(I : integer; procedure P);

    procedure B;
    begin
        writeln(I);
    end;

begin (* A *)
    if I > 1 then
        P
    else
        A(2, B);
end;

procedure C; begin end;

begin (* main *)
    A(1, C);
end.
```



When B is called via formal parameter P, two instances of I exist. Because the closure for P was created in the initial invocation of A, B's static link (solid arrow) points to the frame of that earlier invocation. B uses that invocation's instance of I in its writeln statement, and the output is a 1. With **shallow binding** it would print 2.

Denotational semantics for **deep binding** with **static scoping**

Syntax ***like before***

Procedures have at most one parameter, which is a procedure name

Decl ::= … | **proc** Ide {Com} | **proc** Ide (Ide) {Com} *// Declaration*

Com ::= … | {Decl; Com} | **call** Ide | **call** Ide (Ide) *// Block, invocation*

Semantic domains

Procedures without parameters

Proc0 = **Store → Store**

Procedures with one proc parame

Proc1 = **Proc0 → Store → Store**

Dval = … + Proc0 + Proc1…

Semantic interpretation function

D: Decl → Env → Store → (Env x S

C: Cmd → Env → Store → Store

Semantics: no parameter, static scoping

$D\{\mathbf{proc}\ p\{c\}\}\ r\ s = (r[\alpha_0/p], s)$ **recursion**

where $\alpha_0 = \mu\,\alpha\,.\,C\{c\}\ r[\alpha/p]$

$C\{\mathbf{call}\ p\}\ r = (r(p)$ as **Proc0**$)$

Semantics: one procedural parameter

$D\{\mathbf{proc}\ p(q)\{c\}\}\ r\ s = (r[\alpha_0/p], s)$

where $\alpha_0 = \mu\,\alpha\,.\,\lambda d.C\{c\}\ r[d/q][\alpha/p]$

Deep binding

$C\{\mathbf{call}\ p(h)\}\ r = (r(p)$ as Proc1$)\ (r(h)$ as Proc0$)$

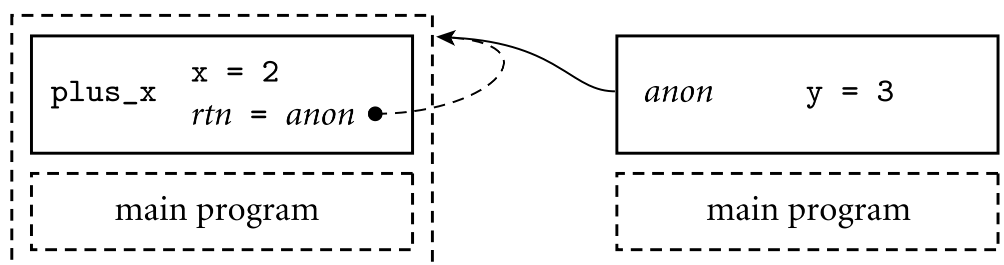
Shallow binding

Requires redefinition of semantic domains

Returning subroutines

- In languages with first-class subroutines, a function **f** may declare a subroutine **g**, returning it as result
- Subroutine **g** may have non-local references to local objects of **f**. Therefore:
  - **g** has to be returned as a *closure*
  - the activation record of **f** cannot be deallocated

```
(define plus-x (lambda (x)
    (lambda (y) (+ x y))))
...
(let ((f (plus-x 2)))
    (f 3))          ; returns 5
```



- **`(plus-x 2)`** returns an **anonymous function** which refers to the local **x**

First-Class Subroutine Implementations

- In functional languages, local objects have *unlimited extent*: their lifetime continue indefinitely
  - Local objects are allocated on the heap
  - *Garbage collection* will eventually remove unused objects
- In imperative languages, local objects have *limited extent* with stack allocation
- To avoid the problem of dangling references, alternative mechanisms are used:
  - C, C++, and Java: no nested subroutine scopes
  - Modula-2: only outermost routines are first-class
  - Ada 95 "containment rule": can return an inner subroutine under certain conditions

Object closures

- Closures (i.e. subroutine + non-local enviroment) are needed only when subroutines can be nested
- Object-oriented languages without nested subroutines can use objects to implement a form of closure
  - a method plays the role of the subroutine
  - instance variables provide the non-local environment
- Objects playing the role of a function + non-local enviroment are called **object closures** or **function objects**
- Ad-hoc syntax in some languages
  - In C++ an object of a class that overrides **operator()** can be called with functional syntax

Object closures in Java and C++

```
interface IntFunc {                              //Java
       public int call(int i);
   }
   class PlusX implements IntFunc {
       final int x;
       PlusX(int n) { x = n; }
       public int call(int i) { return i + x; }
   }
   ...
   IntFunc f = new PlusX(2);
   System.out.println(f.call(3));      // prints 5
```

```
class int_func {                             // C++
   public:
       virtual int operator()(int i) = 0;
   };
   class plus_x : public int_func {
       const int x;
   public:
       plus_x(int n) : x(n) { }
       virtual int operator()(int i) { return i + x; }
   };
   ...
   plus_x f(2);            // f is an instance of plus_x
   cout << f(3) << "\n";                // prints 5
```