Scoping, Functions and Parameter Passing

Programming Language Design 2021

Fritz Henglein
Based on slides by Andrzej Filinski and Torben Mogensen

February 22nd, 2021

Today's lecture

- Main topics:
  - Identifiers and scoping
    - Static vs. dynamic scope
  - Functions
    - Closures
  - Parameter passing
    - "Call by X"
    - Lazy evaluation
- We cover only fundamental concepts of scoping and parameter passing for programming languages in this lecture.
- Please carefully study Chapter 6 of *PLDI* to cover all relevant material there, specifically implementation techniques.

Identifiers

- A programming language typically has several different *classes* of **identifiers**
  - Constants, variables, function names, type names, classes, modules, data constructors, field names, labels, exception names, ...
  - Identifiers may be further classified by types.
- Occurrences of an identifier:
  - *Binding* occurrence; e.g. in local variable or parameter declaration.
  - *Applied* occurrence, an occurrence that
    - either refers statically to a binding occurrence of the same identifier (bound occurrence)
    - or does not have a binding occurrence in the program component to refer to yet (free occurrence)
  - The *same* occurrence may be free within one code fragment, but bound in a larger one.

Scoping

- Scoping: Rules for finding the binding reference of a bound occurrence
- Example from C:

```
void f(int y) {  // 1st binding occurrence of y
   int y = 0;    // 2nd binding occurrence of y
   y = y + 1; }   // two bound occurrences of y
```

  - The two bound occurrences of `y` refer to the second binding occurrence of `y`.
  - The 2nd binding occurrence of `y` has no bound occurrence.
- General *static scoping* principle:
  - Each binding occurrence has a *scope*, a delimited *block* of code containing it. Blocks are nested.
  - A bound occurrence refers to a single binding occurrence in the innermost block that contains both the bound and the binding occurrence.
  - The class and type of the occurrences must match.

Declarations vs. definitions

- *Declaration* (think ":"): introduces an identifier (binding occurrence), often with some intrinsic properties (e.g. type).
  - Examples: formal function parameters, C function prototype, interface declaration in Java-like languages, etc.
- *Definition* (think "="): declaration *plus* immutable binding to particular value/meaning for the identifier.
  - Examples: named function definition, let-bound variable in functional PLs, etc.
  - Note: In most imperative languages, local variable declaration is a *definition* that binds the variable to a *pointer* (address of a mutable memory block), *not* the cell content itself.
- Declared identifiers may be bound to values/meanings :
  - Immediately (in a definition)
  - At compile time (in the same compilation unit)
  - At linking time (for modular programs)
  - At run time (e.g., function parameters)
- Declaration versus definition: often confused!

Simultaneous vs. sequential definitions

- Key question: What is the scope of a definition?
- ```
  let x = 2
  in let y = x + 3    // which definition of x?
         x = 7
     in y             // 5 or 10?
  ```
- Does scope of definition include (recursive definition) or exclude (nonrecursive definition) the right-hand side?

  ```
  let f(x) = x + 1
  in let f(x) = 2 * f(x) // which definition of f?
     in f(5)             // 12 or infinite recursion?
  ```

  - Different conventions across languages, sometimes both forms: `let rec f x = ...` vs. `let f x = ...` in OCaml; `let` vs. `letrec` in Scheme.
  - Often also within a language for different kinds of identifiers, e.g., variable definitions vs. function definitions in C.
  ```

Static vs. dynamic scoping

- Free variable in function: Variable with free occurrence in function definition.
- *Static* scoping (lexical scoping): Free variable(s) refer to *textually* enclosing declaration(s).
- *Dynamic* scoping: Free variable(s) refer to variable binding(s) *call site*.
  - May refer to different definitions in different calls.
  - May fail to find matching definition at run time.
  - May refer to declaration with wrong type at run time.

Static vs. dynamic scoping: Example

```
let y = 2
    f(x) = x + y    // y is free in definition of f
    g(y) = f(y)
in g(5)
```

- Static scoping: Look at program text.
  - `y` in body of `f` is bound to 2.
  - `g(5)` is `f(5)` with `y = 2`. Result: 7.
- Dynamic scoping: Look at run-time environment at call of `f`.
  - `f(y)` is called in the body of g, `y` is bound to 5.
  - `g(5)` is `f(5)` with `y = 5`. Result: 10.

Dynamic scoping: Deep binding

- *Environment*: Active variable bindings (definitions) during evaluation. How to implement?
- *Deep binding*: Stack of local environments, one for each scope entered.
  - Push local environment when entering scope.
  - Pop stack when leaving scope.
  - Lookup: Variable looked up in top of the stack (current scope); if not found, in next local environment (enclosing scope), etc.
- Fast entering and leaving of scopes; slow lookup for nonlocal variables.
- Simplified: stack of bindings; lookup: find topmost binding for identifier.

Dynamic scoping: Shallow binding

- *Environment*: Active variable bindings (definitions) during evaluation. How to implement?
- *Shallow binding*: Current environment for active bindings, plus stack of hidden bindings.
  - When entering scope, for each new binding
    - push current binding for same variable to hidden bindings and
    - update current environment with new binding.
  - When leaving scope, restore previous bindings in current environment by popping them from hidden bindings.
  - Lookup: Look variable up in current environment.
- Fast lookup for all variables. Slow entering and leaving of scopes with many definitions.

Static scoping: Nonnested functions

- Consider definitions:

```
int y;
int f(int x) { int t = x + 1;  return t * y; }
int m(int z) { y = 10; return f(2*y) + z; }
```

- At run time, allocate new *activation record* for `f`:
  - Space for *result* (will be assigned 210).
  - Space for *parameters* (`x`, initialized to 20 in `m`).
  - *Code pointer (return address)* to code point in caller to jump to when returning, "`return ... + z;`")
  - *Frame pointer (dynamic link)* to caller's activation record (containing, e.g, `z`)
  - Space for *local variables* (`t`, will be assigned 21).
- `y` is global var at fixed address, so not part of record.
- Activation records normally allocated on a *stack* with *stack pointer* pointing to first available memory cell.

Nested (local) functions

- Function defined in body of another function.
  - Declaration invisible outside outer function.
  - May have free variables that reference declarations in outer function (static scoping).
- Language without nested functions: C.
- Languages with nested functions: C with local functions (e.g., gcc), Haskell, Python, Java, etc.
- Example:

```
int m(int z) {
  int y = 10;
  int f(int x) {int t = x+1; return t * y;}
  return f(2*y) + z;
}
```

  - Body of `f` also needs access to the value of *stack*-allocated `y` from the outer scope.

Nested (local) functions: Implementation

- Make free variables extra, implicit parameters of `f`.
  - If `f` can modify free variables, should be passed by reference (see later).
- Pass stack activation record (frame) of `m` as extra argument to `f` (*static link*).
  - Chain or record of links for deeper nesting (Algol-like language)
  - Works only if inner function is called before outer function returns.
- Function closures: function pointer plus heap-allocated record with bindings to free identifiers
  - Works also after outer function has returned.
- λ-lifting: Move nested function definitions to top-level.
- Defunctionalisation: Remove function pointers of λ-lifted code.

Nested (local) functions: Example

Nested, mutually recursive functions `f`, `g`, `h`.

```
fun f x =
  let val a = x + 1
      fun g y =
        let val b = x - 1
            fun h z = if z = 0 then b + x else f b + a
        in h y + h b end
  in g a + 3 end
```

- `f` is closed (has no free variables).
- `g` has free variables `x`, `a`.
- `h` has free variables `x`, `a`, `b`.

λ-lifting

Transform non-closed functions into closed functions by adding parameters for their free variables.

```
fun f x =
  let val a = x + 1
      fun g y =
        let val b = x - 1
            fun h z = if z = 0 then b + x else f b + a
        in h y + h b end
  in g a + 3 end
```

λ-lifted:

```
fun f'       x = let val a = x + 1 in g' x a a + 3 end
fun g' x a   y = let val b = x - 1 in h' x a b y + h' x a b b
fun h' x a b z = if z = 0 then b + x else f' b + a
```

Closure form

λ-lifted:

```
fun f'        x = let val a = x + 1 in g' x a a + 3 end
fun g' x a    y = let val b = x - 1 in h' x a b y + h' x a b b
fun h' x a b z = if z = 0 then b + x else f' b + a
```

Closure form: grouping parameters into implicit (λ-lifted) and explicit (given):

```
fun f' ()        x = let val a = x+1 in g' (x, a) a + 3 end
fun g' (x, a)    y = let val b = x-1 in h' (x, a, b) y + h' (x, a, b) b
fun h' (x, a, b) z = if z = 0 then b + x else f' () b + a
```

All functions are closed and fully applied (no partial applications).

- Can be implemented as function pointers
- Arguments are passed by parameter passing method as given in programming language (see below)
- Closure form: Exactly 2 parameters.

Higher-order functions (first-class functions)

Combinator (closed function): Function with no free variable occurrences, typically implemented as *function pointer* (address of code of its body).

First-order function: Function that neither accepts functions as arguments nor returns them

Higher-order function: Function that is not a first-order function

Challenge: Passing *non-closed* functions as argument/result to/from higher-order function.

Higher-order functions

- In general, a function value is represented as a *(function) closure* at run time:
  1. A code pointer (like in C)
     - Or just a code *index* for a *dispatcher* function containing all function bodies (*defunctionalisation*)
  2. Environment for free variables in function definition.
- "Downwards funarg": allow only passing closure to another function
  - Examples: Algol 60, Pascal, ...
  - Static link: Pointer to activation record with bindings for free variables. on the stack.
- "Upwards funarg": also allow returning from another function
  - Scheme, ML, ...
  - Activation records referenced by a closure may have been be deallocated before closure is invoked.
    - Must heap allocate (at least some) free variables on heap
  - Allows closures to be stored in general data structures: *first-class functions*.

Closures as partial function applications

- *Curried function*: Function that can be applied to one argument at a time: $f\,x\,y = x + y$ (instead of $f(x, y) = x + y$).
- *Partial application*: Result of applying curried function to one argument at a time, e.g. $f\,5\,8$.
- *Function closure*: Closed curried function of two arguments, applied to first argument, e.g. $f\,5$. Represented at run time as a pair consisting of f (function pointer) and argument 5, the value of its first argument.

Closure conversion for higher-order functions

Example of higher-order function:

```
f (x, p) = p x + p 1
g a = let fun h c = a + c
      in (f (a, h) + f (17, g), h)
```

Note: `h` is not closed.
After λ-lifting:

```
f'   (x, p) = p x + p 1
g'   a      = (f' (a, h' a) + f' (17, g'), h' a)
h' a c      = a + c
```

Note: `h' a` is a partial application.

Closure conversion for higher-order functions

After λ-lifting:

```
f'    (x, p) = p x + p 1
g'    a      = (f' (a, h' a) + f' (17, g'), h' a)
h' a c       = a + c
```

After conversion of partial applications to closure form:

```
f' () (x, p) = p x + p 1
g' () a      = (f'() (a, h' a) + f'() (17, g'()), h' a)
h' a  c      = a + c
```

Note:

- All functions have two arguments, one for implicit parameter(s), one for explicit parameter(s).
- All data passed are first-order values or function closures.

Defunctionalisation

- Goal: Implement higher-order functions without function pointers, using first-order values only
- Idea: Instead of passing a function pointer pass the *name* of the function.
  - Replace function pointer `f` in a function closure by a unique constructor `F`
  - Define evaluation function `eval` such that

    ```
    eval (F x) y = f x y
    ```

Defunctionalisation: Example

Starting point: Closure-converted program.

```
f' () (x, p) = p x + p 1
g' () a      = (f'() (a, h' a) + f'() (17, g'()), h' a)
h' a  c      = a + c
```

Step 1: Define function `eval` that maps constructors to the top-level functions such that

```
eval (F ()) (x, p) = f' () (x, p)
eval (G ()) a      = g' () a
eval (H a)  c      = h' a c
```

We get

```
eval (F ()) (x, p) = p x + p 1
eval (G ()) a    = (f'() (a, h' a) + f'() (17, g'()), h' a)
eval (H a)  c      = a + c
```

Defunctionalisation: Example

```
eval (F ()) (x, p) = p x + p 1
eval (G ()) a    = (f'() (a, h' a) + f'() (17, g'()), h' a)
eval (H a)  c       = a + c
```

Step 2: Replace function pointers in closure *constructions* by their names

```
eval (F ()) (x, p) = p x + p 1
eval (G ()) a    = (F () (a, H a) + F () (17, G ()), H a)
eval (H a)  c       = a + c
```

and add the missing `eval` at closure *applications*:

```
eval (F ()) (x, p) = eval p x + eval p 1
eval (G ()) a       = (eval (F ()) (a, H a) +
                        eval (F ()) (17, G ()), H a)
eval (H a)  c       = a + c
```

Defunctionalisation: Example

Defunctionalised code:

```
eval (F ()) (x, p) = eval p x + eval p 1
eval (G ()) a      = (eval (F ()) (a, H a) +
                      eval (F ()) (17, G ()), H a)
eval (H a)  c      = a + c
```

- Only one function, which is first order: `eval`.
  - `eval` is called `dispatch` in lecture notes.
- No function closures, no function pointers.
- Requires whole program analysis: Find all top-level functions that can be passed to another function.
- `eval` can be statically typed with generalized algebraic data types, but not ordinary data types.

Scoping and first-class functions: Summary

- Binding and applied occurrences of variables
- Static scoping vs. dynamic scoping: How and when free variables in functions are bound
- Nested function: Statically scoped function with free variables.
- Function closure: Closed function (function pointer) plus (heap-allocated) bindings for its free variables
  - Optimizations: No free variables (function pointer only); downargs only (bindings on stack okay); bindings by deep binding, shallow binding or combination thereof; bindings split into those on stack, those on heap; etc.
  - General model: Function closure = closed curried function of two arguments applied to one argument.
- Defunctionalisation: Replacing function pointers by function names (constructors).

Parameter passing methods

- How are argument values passed from caller of procedure to callee? And how are results returned?
- Minimal solution: through shared global variables.
  - Sometimes only possibility (COBOL `PERFORM`, BASIC `GOSUB`)
  - Or may be used as supplement to other methods
- Great variability across programming languages.
  - Boils down to relatively few underlying concepts
  - Often several forms available in single language; programmer selects most appropriate for each purpose.
- Fundamental distinction: are arguments to function evaluated by *caller* (proactively) or *callee* (only when used)?
  - Eager evaluation versus "lazy" evaluation.
  - Will look at each in turn.

Eager approach: Call by value

- Function definition header includes list of *(formal) parameters*, possibly typed.
- At call site, *actual parameters* (aka. *arguments*) may be general expressions of appropriate types
- Arguments evaluated, and formal parameters bound/initialized to their values, before execution of callee body.
  - In language with side effects, order of arguments may matter.
  - Depends on language whether evaluation order specified.
- Not immediately possible to pass results back.
  - Even if argument is a mutable variable, assignments to formal parameter in body will not modify argument.
  - But if *address* of mutable object/variable (i.e., pointer) is passed by value, callee can still modify *content* of passed memory.
- In general, need to be careful about deep vs. shallow copy (e.g., C passes structs by value, but arrays by pointer)

Call by value–result

- Aka. copy-in/copy-out, call-by-value-return.
  - Special case: Call by copy out, for uninitialized actual parameters
- Actual parameter must be a variable (more generally: l-value, e.g., `x[i]`), not a general expression.
  - But identity of variable still determined eagerly by caller, e.g., in `p(x[i+1])`, index expression `i+1` evaluated *before* the call.
- Values copied from actual to formal parameters before procedure body is executed, and from formals to actuals when returning.
- In general, order of writebacks may matter.
  - Language may require that actual parameters are *distinct* variables.
  - But this is hard to enforce statically, e.g., `p(x[i], x[j])`.

Call by reference

- Syntax looks like call by value–result (actual parameter must be an l-value)
  - But semantics corresponds to passing *address* of actual parameter by value, and making all reads/writes of formal parameter indirect.
- For two distinct purposes
  - Avoiding copying of large values
  - Allowing bidirectional data transfer
  - Not always evident which is intended!
- Assignments to formal parameter *immediately* reflected in actual parameter.
  - Makes observable difference if same actual parameter variable used for two distinct formal parameters (or is also globally accessible).

Call by reference, continued

- Fortran: Only call-by-reference, but allows actual parameter to be general expression, e.g., `X+3`.
- Nominal semantics: caller creates *anonymous* local variable for actual parameter, initializes to value of expression.
  - So any assignments to parameter in callee have no net effect.
- Natural "optimization": if argument is a constant (literal), just pass address of constant in program code.
  - Often preferable to allocating space for extra variable and generating code for initializing it
  - With $\sim 2^{16}$-bytes address spaces, every word counts!
- Also: if same constant used multiple times in program, just keep a single copy ("literal pool")
  - What could possibly go wrong...?
- Cf. "Hacker purity test" (ca. 1989):

```
0015 Ever change the value of 4?
0016 ... Unintentionally?
0017 ... In a language other than Fortran?
```

Lazy approaches: call by text/macro expansion

- Argument expressions passed verbatim to callee without any interpretation by caller.
- Found in, e.g., TEX/LATEX, Lisp `flambda`, many scripting languages (Bash, Tcl, ...)
- Requires interpretation (or even string parsing) at run time.
  - Effectively forces dynamic scope.
  - Poorly suited for compilation, or even high-performance interpretation.
  - Unpredictable semantics, error prone, extremely insecure (injection attacks, cross-site scripting attacks, passing arguments that arbitrarily update local variables).

Call by name

- Sanitized version of call by text.
  - Statically scoped.
  - Capture avoiding substitution.

- Relatively easy to specify formally by Algol "copy rule":
  1. Replace formal parameters in procedure body with actual parameters
     - Respecting syntactic structure and types, cf. `let p(x)=2*x in p(3+y)`
     - Performing capture-avoiding substitution (as in logic, λ-calculus, etc)
  2. Replace procedure call with above-modified procedure body

```
procedure p(x, r); integer x, r;
  begin integer y; y := x+2; r := x*y end;
integer y;
...
p(y-3, y)  "="
begin integer y1; y1:= (y-3)+2; y := (y-3)*y1 end
```

Call by name, continued

- Copy rule (β-equality) easy to explain, used for inlining in optimizing compilers at compile time.
- How to implement at run time?
- Parameter expression passed as (compiled) parameterless function ("thunk"), to be invoked by callee when value needed.
  - Plus a "setter" procedure for assignments to parameter.
  - For non-variable actual parameters, setter signals runtime error.
    - Cf. Fortran behavior above.
    - Not enough information to detect problem at compile time.
- Default parameter passing mechanism in Algol 60:
  - Call by value formally defined in terms of declaring extra local variable, initialized to value of parameter passed by name.
  - In practice, has also call by value (`value k; integer k`).
- Call by name with mutable variables allows some nasty code.
  - E.g., "Jensen's Device": `p(x[i])` where callee may modify `i`.

Call by need

- In a purely functional language (no side effects), evaluating an expression twice always yields the same result.
    - Where a "result" may also be a runtime error (e.g., division by zero) or nontermination.
- Observation: for called-by-name parameter, can *cache* any successful result of first evaluation, and just return that on all subsequent evaluations. No setter required.
- Normally implemented as destructive update: thunk contains mutable cell.
    - But update is completely transparent to programmer: observable behavior is still like call by name, only faster.
- Known as "call by need"; best of both worlds:
    - If formal parameter never used, actual never evaluated (like for call by name).
    - If formal parameter used multiple times, actual only evaluated once (like for call by value).

General lazy evaluation

- Can generalize from parameter passing to definitions in general, also for data structures:

```
let xs = [1+2, 3+4, 5+6]
    x = head (tail xs)
in  x*x
```

- Returns 49, but only evaluates 3+4, and only once.
- In general, evaluation is *graph reduction* of expression DAG
  - Graphs may even have cycles, in case of recursion.

Properties of parameter passing methods

- Let f be function with body $E = \ldots x \ldots y \ldots$ containing free occurrences of x, y: $f(x, y) = E[x, y]$
- Substitution: Write $E[u/x, v/y]$ for E where all free occurrences of x and y are replaced by u and v, respectively.
  - Substitution must be *capture avoiding*: No free variable in u or v must become bound.

Properties:

Call by reference: $f(u, v) \cong E[u/x, v/y]$

Call by value: $f(e_1, e_2) \cong x := e_1, y := e_2; E[x, y]$

Call by value-return: $f(u, v) \cong x := u, y := v; r := E[x, y]; u := x, v := y; r$

Call by name: $f(e_1, e_2) \cong E[e_1/x, e_2/y]$

Call by need: $f(e_1, e_2) \cong E[e_1/x, e_2/y]$
Operationally, replace e_i by its value v_i if and when the value of e_i is *needed the first time*.

Parameter passing methods: Summary

- Call by value: Evaluate argument expression and bind or assign its value to formal parameter in function.
- Call by name: Substitute (capture-avoiding) argument expression for formal parameter in function.
  - Call by need: Optimized version of call by name; if value of expression is always the same (no side effects).
  - Not applicable to Algol: Has side effects (Jensen's device).
- Call by reference: Alias (l-value of) argument expression with formal parameter in function; they are the same variable.
  - Cannot rely on distinct identifiers in function to be distinct variables.
- Call by value-result: Evaluate (l-value of) argument expression and assign it to formal parameter in function; and value of formal parameter back to argument expression at end.
  - Similar to call by reference, but with no aliasing of formal parameters during evaluation of function body.