Subroutines and Control Abstraction

Textbook, Chapter 8

Mechanisms for process abstraction

- Single entry (except FORTRAN, PL/I)
- Caller is suspended
- Control returns to caller

Design Issues

- Syntax, type checking
- Parameter passing mechanism
- Static or dynamic allocation of locals
- Static or dynamic scope
- Environment of parameter functions
- Overloading
- Generics
- Separate compilation
Syntax

- **FORTRAN**
  
  SUBROUTINE ADDER (parameters)

- **Ada**
  
  procedure ADDER (parameters)

- **C**
  
  void ADDER (parameters)

More Syntax

- Positional or keyword parameters
- Default values

**Ada example:**

```ada
function F(X : FLOAT;
  Y : INTEGER := 1;
  Z : FLOAT) RETURN FLOAT;

I := F(3.14, Z => 2.71);
```

Type Checking

- Are argument types checked against parameter types?
- If functions are passed as arguments, is the function type checked?

  - Kernighan & Richie C:
    
    ```c
    int foo(int (*f)()) { ... }
    int bar(int i, int j) { ... }
    foo(bar);
    ```
Fixed-point Version of Factorial

- In K&R C, can’t be typed in ANSI C

```c
int f(int (*g)(), int n) {
    if (n == 0)
        return 1;
    else
        return n * g(g, n-1);
}

int fac(int n) {
    return f(f, n);
}
```

Parameter-Passing Methods

- Call-by-Value (Copy-In, Eager Eval.)
- Call-by-Result (Copy-Out)
- Copy-In-Copy-Out
- Call-by-Reference
- Call-by-Name
- Call-by-Need (Lazy Evaluation)

Call-by-Value

- Allocate memory for parameter
- Evaluate argument
- Initialize parameter with value
- Call procedure
- Nothing happens on return

- Used in C, C++, Pascal, Lisp, ML, etc.
Call-by-Value Example

```c
int a = 1;
void foo(int x) {
    // a and x have same value
    // changes to a or x don’t
    // affect other variable
}

// argument can be expression
foo(a + a); foo(a);
// no modifications to a
```

Call-by-Result

- Argument must be variable
- Allocate memory for parameter
- Don’t initialize parameter
- Call procedure
- ...
- Copy parameter into argument var.
- Return from procedure

Call-by-Result Example

```c
int a = 2;
void foo(int x) {
    // x is not initialized
    // changes to a or x don’t
    // affect other variable
}

// argument must be variable
foo(a);
// a was modified
```
Call-by-Value-Result
(Copy-In-Copy-Out)

- Combination of previous two
- Copy argument value on call
- Copy result on return
- Used by Ada for parameters of primitive type in in-out mode

Call-by-Value-Result Example

```c
int a = 3;
void foo(int x) {
    // a and x have same value
    // changes to a or x don't
    // affect each other
}
// argument must be variable
foo(a);
// a might be modified
```

Call-by-Reference

- Evaluate argument into temporary
- Parameter is alias for location of tmp
- Call procedure
- Nothing happens on return
- Used by FORTRAN before 1977, Pascal var parameter
Call-by-Reference Example

```c
int a = 4;
void foo(int x) {
    // a and x reference same location
    // changes to a or x
    // affect each other
}
// argument can be an expression
foo(a + a); foo(a);
// a might be modified
```

Call-by-Ref in FORTRAN IV Implementations

```fortran
SUBROUTINE FOO(I)
I = 5
RETURN

FOO(10)
J = 10

J gets value 5!
```

Call-by-Name

- Don’t evaluate argument
- Create closure to evaluate argument
- Call procedure
- Eval arg by calling parameter closure
- Nothing happens on return

- Used by ALGOL-60, Simula-67
- Similar to macro expansion (e.g., TeX)
Call-by-Name Example

```c
int a = 5;
void foo(int x) {
    // x is a function
    // to get value of argument,
    // evaluate x() when value is needed
}
// argument can be an expression
foo(a + a);  foo(a);
// no modifications to a
```

---

Call-by-Need (Lazy Evaluation)

- Similar to Call-by-Name
- Argument evaluated only once
- Result kept in temporary
- Behavior differs with side effects
- Used by Haskell, Miranda

---

Example Lazy Evaluation

```c
int diverge() {
    return diverge();
}
int if_then_else(int c, int t, int e) {
    return c ? t : e;
}
int i = if_then_else(1, 42, diverge());
```
Static vs. Dynamic Locals

- **Static**
  - efficient
  - no time for (de)allocation required
  - retain value across function calls
- **Dynamic (on the stack)**
  - required for recursive functions
  - on modern hardware cheap access
  - let compiler do the optimizations

Implementation of Parameter Passing

- Allocate activation record on heap, in registers, or on the stack
- Copy values into activation record
- Copy pointers for Call-by-Reference
- Encapsulate lazy args. in closure
- Branch to function code
- Standard calling convention: stack

Environment of Function Parameters

- Function is passed down
  ```
  int x = 1;    // static scoping
  int foo() { return x; }
  int bar(int (*f) ()) {
    int x = 2;  // dynamic scoping
    return f();
  }
  int i = bar(foo);
  ```
Static vs. Dynamic Scope

- **Static scope**
  - non-local variables are looked up in statically enclosing scope
  - Need static link for nested functions
  - i gets value 1
- **Dynamic scope**
  - non-locals are looked up along dynamic call chain
  - i gets value 2

Environment of Function Parameters

- Function is passed up
  
  ```scheme
  (define (add n)
    (lambda (m) (+ n m)))
  ```

  ```scheme
  (define add1 (add 1))
  (define add5 (add 5))
  (define i (add1 10))
  (define j (add5 10))
  ```

Implementation of Closures

- Functions are represented as pair
  - pointer to code
  - pointer to environment
- Activation records might be on heap
- Needs garbage collection
  - Programmer doesn't know when/how to reclaim the activation record of add
Garbage Collection

- No explicit free or delete
- When memory runs out, GC
  - traverses life data structures starting with global variables and stack
  - marks reachable data
  - scans heap to find unreachable data
  - collects unreachable data in free list
  - may compact heap

Heap Data Structure

- For manual allocation, copying GC
  - Blocks of storage with free list
  - Each storage block contains length
  - Free blocks are linked into free list
  - First Fit: allocate object in first block that's large enough, keep rest on free list
  - Combine blocks when deleting objects

Heap Data Structure (cont’d)

- For Copying GC
  - Only needs top of heap pointer

- Other possibilities (less efficient)
  - Linked list with Best Fit
  - Binary tree
GC Algorithms

- Mark & Sweep style
  - Traverse and mark all live data structures
  - Sweep heap, unmarked data on free list
  - For C/C++: conservative GC

- Copying GC
  - Allocate new heap (one page at a time)
  - Copy objects to new heap, compact
  - Update pointers

GC Algorithms (cont’d)

- Generational GC
  - New objects become garbage quicker
  - Frequently collect new generation
  - Rarely collect old generation

- Modern GC, e.g., in Java VM
  - Generational GC
  - Mark & sweep for young generation
  - Copying for old generation

Conservative GC for C/C++

- Pointers and integers look alike
  - Copying collection doesn’t work
  - Integers are treated as potential pointers
  - May not find all garbage but is safe
  - May not work with pointer arithmetic
    - free() is no-op, malloc() runs GC
    - Boehm-Demers-Weiser GC
    - [http://www.hboehm.info/gc/](http://www.hboehm.info/gc/)
Overloading vs. Multimethods

- Multiple methods/functions with same name, different arg. types
  ```
  int foo(int);
  int foo(float);
  ```
- Overloading
  - method selection at compile time
- Multimethods
  - method selection at run time

Overloaded Operators

- Allow redefining operators
- Ada
  - function "**" ...
- C++
  - operator *
  - allows redefinition of [], ->, (), etc.

Generic Functions

- Functions with type parameter
  ```
  template <class Type>
  Type max (Type n, Type m) {
    return n > m ? n : m;
  }
  ```
- Instantiated at compile time
- Better than using preprocessor
Separate Compilation

- Compiler needs type information of other compilation units
- C++
  - Use #include
  - Split code into header and source file
- Java
  - Search in current directory
  - Compile other classes if needed