CSE 505 Lecture Notes:

Parameter Passing

September 1999

Introduction

techniques used for argument passing:

- call by value
- call by result
- call by value-result
- call by reference
- call by name

(and call-by-constraint in constraint languages)

call by value: copy going into the procedure

call by result: copy going out of the procedure

call by value result: copy going in, and again going out

call by reference: pass a pointer to the actual parameter, and indirect through the pointer

call by name: re-evaluate the actual parameter on every use. For actual parameters that are simple variables, this is the same as call by reference. For actual parameters that are expressions, the expression is re-evaluated on each access. It should be a runtime error to assign into a formal parameter passed by name, if the actual parameter is an expression. Implementation: use anonymous function ("thunk") for call by name expressions

efficiency of call by reference -- large arrays

Fortran uses call by reference
insecurities in early FORTRANs -- passing a constant

Algol 60 has call by name, call by value

Ada uses different designations: `IN`, `OUT`, `IN OUT`:

- For scalar data types (such as integers), `IN` is the same as call by value, `OUT` is the same as call by result, and `IN OUT` is the same as call by value result. In addition, `IN` parameters are local constants -- you can't assign into them.
- For compound data types (such as arrays), these can be implemented as above, or using call by reference. (You can write a test program to determine which method your compiler is using -- however, programs that rely on one implementation choice or the other are "erroneous".)

Lisp and Smalltalk use call-by-value with pointer semantics. Java uses call-by-value -- with just copying for primitive types, and pointer semantics for objects.

An important related concept: **aliasing**. Two variables are aliased if they refer to the same storage location. A common way that aliasing can arise is using call by reference. A formal parameter can be aliased to a nonlocal variable, or two formal parameters can be aliased.

In a functional programming language, call by value is equivalent to applicative order evaluation. Call by name is equivalent to normal order evaluation. (It always gives the same results as lazy evaluation, but lazy evaluation may be faster.)

Example 1: illustrates call by value, value-result, reference

```
begin
integer n;
procedure p(k: integer);
    begin
    n := n+1;
    k := k+4;
    print(n);
    end;
n := 0;
p(n);
print(n);
end;
```

Note that when using call by reference, `n` and `k` are aliased.

Output:

```
call by value:        1 1
call by value-result: 1 4
call by reference:    5 5
```

Example 2: Call by value and call by name

```
    begin
    integer n;
    procedure p(k: integer);
        begin
        print(k);
        n := n+1;
        print(k);
        end;
    n := 0;
    p(n+10);
    end;
```

Output:

```
call by value:    10 10
call by name:     10 11
```

Example 3: Call by value and call by name (with evaluation errors)

```
    begin
    integer n;
    procedure p(k: integer);
        begin
        print(n);
        end;
    n := 5;
    p(n/0);
    end;
```

Output:

```
call by value:    divide by zero error
call by name:     5
```

Example 4: Non-local references

```
  procedure clam(n: integer);
  begin

    procedure squid;
    begin
      print("in procedure squid -- n="); print(n);
    end;

    if n<10 then clam(n+1) else squid;

  end;

clam(1);
```

Output:

```
in procedure squid -- n=10
```

Example 5: Procedures as parameters

To pass a procedure as a parameter, the system passes a *closure*: a reference to the procedure body along with a pointer to the environment of definition of the procedure.

```
  begin

  procedure whale(n: integer, p: procedure);
  begin

    procedure barnacle;
    begin
      print("in procedure barnacle -- n="); print(n);
    end;

    print("in procedure whale -- n="); print(n);
    p;
    if n<10 then
      begin
        if n=3 then whale(n+1,barnacle) else whale(n+1,p)
      end
  end;

  procedure limpet;
  begin
    print("in procedure limpet");
  end;


whale(1,limpet);

end;
```

Output:

```
 in procedure whale -- n=1
 in procedure limpet

 in procedure whale -- n=2
 in procedure limpet

 in procedure whale -- n=3
 in procedure limpet

 in procedure whale -- n=4
 in procedure barnacle -- n=3

 in procedure whale -- n=5
 in procedure barnacle -- n=3

 in procedure whale -- n=6
 in procedure barnacle -- n=3

 in procedure whale -- n=7
 in procedure barnacle -- n=3

 in procedure whale -- n=8
 in procedure barnacle -- n=3

 in procedure whale -- n=9
 in procedure barnacle -- n=3

 in procedure whale -- n=10
 in procedure barnacle -- n=3
```

Similar effects can be achieved by passing labels as parameters.