CS403: Programming Languages

Notes on Invariants

Description

There are six main methods of passing arguments in a function call:

- call-by-value
- call-by-reference
- call-by-value-result
- call-by-name
- call-by-need
- call-by-name-with-thunks

Let's examine these six methods in turn.

Call-by-value

This method is the only method of parameter passing allowed by C, Java, Scheme, and Scam. In this method, the formal parameters are set up as local variables that contain the value of the expressions that were passed as arguments to the function. Changes to local variables are not reflected in the actual arguments. For example, a swap routine such as:

```
function swap(a, b)
    {
    var tmp;
    tmp = a;
    a = b;
    b = tmp;
    }
```

under *call-by-value* would not yield the intended semantics. The code fragment:

```
var x = 3;
var y = 4;

swap(x,y);
println("x is ", x, " and y is ", y);
```

would print out:

```
x is 3 and y is 4
```

since only the values of the local variables a and b were swapped. In general, one cannot write get a swap routine to work under *call-by-value* unless the addresses of the variables are somehow sent. One way of using addresses is to pass an array (in C and Java, when an array name is used as an argument, the address of the first element (C) or array object (Java) is sent, rather than a copy of the array). For example, the code fragment:

```
int x[1];
int y[1];

x[0] = 3;
y[0] = 4;

swap(x,y); //address of beginning element is sent

printf("x[0] is %d and y[0] is %d\n",x[0],y[0]);
```

with *swap* defined as:

```
void swap(int a[], int b[])
    {
    int tmp;

    tmp = a[0];
    a[0] = b[0];
    b[0] = tmp;
    }
```

would print out:

```
x[0] is 4 and y[0] is 3
```

In this case, the address is stored in the local variables a and b. This is still *call-by-value*; even if the address stored in a, for example, is modified, x still "points" to the same array as before.

C has an operator that extracts the address of a variable, the & operator. By using &, one can write a swap in C that does not depend on arrays:

```
void swap(int *a,int *b)
    {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
    }
```

The call to *swap* would look like:

```
int x = 3;
int y = 4;

swap(&x,&y);

printf("x is %d and y is %d\n",x,y);
```

Note that this is still *call-by-value* since the value of the address of x (and y) is being passed to the swapping function.

Call-by-reference

The second method differs from the first in that changes to the formal parameters during execution of the function body are immediately reflected in actual arguments. Both C++ and Pascal allow for *call-by-reference*. Normally, this is accomplished, under the hood, by passing the address of the actual argument (assuming it has an address) rather than the value of the actual argument. References to the analogous formal parameter are translated to references to the memory location stored in the formal parameter. In C++, *swap* could be defined as:

```
void swap(int &a, int &b) //The & here signifies call-by-reference
    {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
    }
```

Now consider the code fragment:

```
var x = 3;  //assume x at memory location 1000
var y = 4;  //assume y at memory location 1008

//location 1000 holds a 3
//location 1008 holds a 4

swap(x,y);
cout << "x is " << x << " and y is " << y << "\n";
```

When the swapping function starts executing, the value 1000 is stored in the local variable a and 1008 is stored in local variable b. The line:

```
tmp = a;
```

is translated, not into store the value of *a* (which is 1000) in variable *tmp*, but rather store the value at memory location 1000 (which is 3) in variable *tmp*. Similar translations are made for the remaining statements in the function body. Thus, the code fragment prints out:

```
x is 4 and y is 3
```

The swap works! When trying to figure out what happens under *call-by-reference*, it is often useful to draw pictures of the various variables and their values and location and update them as the function body executes.
What happens if a literal is passed as an actual argument to a formal parameter that is designated *call-by-reference*? For example, how might the call:

```
swap(x,4);
```

be handled?

One can simulate *call-by-reference* in Scam by delaying the arguments and capturing the calling environment. Using this paradigm, example, a working *swap* routine can be written in Scam:

```
(define (swap # $a $b)
    (define temp (get $a #))
    (set $a (get $b #) #)
    (set $b temp #)
    )
```

The *get* function takes a symbol and an environment and retrieves the value of the symbol as found in the given environment. The *set* function is analogous, but changes the value of a symbol found in the given environment. Both *get* and *set* are true functions, so the values of *$a* and *$b* are used in manipulating the given environment. Note that this version of *swap* only works if symbols are passed to *swap*.

By taking advantage of Scam's allowing us to make calls to *get* implicitly by calling the object itself, *swap* can be rewritten as:

```
(define (swap # $a $b)
    (define temp (# $a))
    (set $a (# $b) #)
    (set $b temp #)
    )
```

Call-by-value-result

This method is a combination of the first two. Execution of the function body proceeds as in *call-by-value*. That is, no updates of the actual arguments are made. However, after execution of the body, but just before the function returns, the actual arguments are updated with the final values of their associated formal parameters. This method of parameter passing is often used for Ada *in-out* parameters.

We can simulate *call-by-value-result* in Scam by simulating *call-by-reference*, but assigning the values of the passed symbols to local variables at the start of the function:

```
(define (swap # $a $b)
    ; pre
    (define a (get $a #))
    (define b (get $b #))

    ; body
    (define temp a)
    (set! a b)
    (set! b temp)

    ;post
    (set $a a #)
    (set $b b #)
    )
```

The section marked verb!pre! sets up the locals while the section marked verb!body! implements the swap on the locals. The section marked verb!post! updates the variables that were passed to *swap*.

Usually, one cannot tell whether a language implements *call-by-reference* or *call-bay-value-result*; the resulting values are the same. One situation where the two methods of parameter passing can generate different results is if the body portion of the function references a global variable and that global variable is passed as an argument as well. This second reference to the global is known as an *alias*. Now there are two references to the same variable through two different names. Unlike *call-by-value-result*, updates to the alias in the body section are immediately reflected in the value of the global variable under *call-by-reference*.

Call-by-name

Call-by-name was used in Algol implementations. In essence, functions are treated as macros. Under *call-by-name*, the fragment:

```
var x = 3;
var y = 4;
swap(x,y);
print("x is ", x, " and y is ", y, "\n");
```

would be translated into:

```
var x = 3;
var y = 4;

//substitute the body of the function for the call,
//renaming the formal parameters with the names of
//the actual args

{
var tmp;

tmp = x;
x = y;
y = tmp;
}

print("x is ", x, " and y is ", y, "\n");
```

Under *call-by-name*, *swap* also works, so why is *call-by-name* a method that has fallen into relative disuse? One reason is complexity. What happens if a local parameter happens to have the same name as one of the actual args. Suppose *swap* had been written as:

```
function swap(a, b)
    {
    var x;

    x = a;
    a = b;
    b = x;
    }
```

Then a naive substitution and renaming would have produced:

```
var x = 3;
var y = 4;

//substitute the body of the function for the call,
//renaming the formal parameters with the names of
//the actual args

{
var x;

x = x;
x = y;
```

```
y = x;
}

println("x is ", x, " and y is ", y);
```

which is surely incorrect. Further problems occur if the body of the function references globals which have been shadowed in the calling function. This requires a complicated renaming scheme. Finally, *call-by-name* makes treating functions as first-class objects problematic (being difficult to recover the static environment of the called function). *Call-by-name* exists today in C++, where it is possible to *inline* function calls for performance reasons.

Call-by-need

In *call-by-value*, the arguments in a function call are evaluated and the results are bound to the formal parameters of the function. In *call-by-need*, the arguments themselves are literally bound to the formal parameters, as in *call-by-name*. A major difference is that the calling environment is also bound to the formal parameters as well. This bundle of literal argument and evaluation environment is known as a *thunk*. The actual values of the arguments are determined only when such values are needed; when such a need occurs, the thunk is evaluated, causing the literal argument in the thunk to be evaluated in the stored (calling) environment. For example, consider this code:

```
function f(x)
    {
    var y = x;      //x needed! x is fixed to its current value
    z = z * 2;
    return x + y;   //x needed! x was already evaluated under call-by-need
    }

var z = 5;
f(z+3);
```

Under *call-by-name*, the return value is 21, but under *call-by-need*, the return value is 16. This is because the value of z changed ***after*** the point when the value of x (really z+3) was needed and the value of x was fixed from then on. Under *call-by-name*, the second reference to x causes a fresh, new evaluation of z, the yielding the result of 21.

Call-by-need is exactly the method used to implement streams in the text. It is important to remember that the evaluation of a *call-by-need* argument is done only once, with the result stored for future requests.

One can simulate *call-by-need* in Scam by delaying the evaluation of the arguments and capturing the calling environment. The combination of delayed argument and captured environment comprise a thunk in Scam. To memoize the thunk, we assign the evaluation of the delayed argument to a local variable. Here is a Scam's version of function *f*, if we assume the argument to *f* is always an identifier:

```
(define (f # $x)
    ;get-and-set the value of x
    (define x (# $x))
    (define y x)
    (set! z (* z 2))
    (+ x y)
    )
```

If we allow *f* to be called with an arbitrary expression, we need a more powerful tool to evaluate the delayed argument, *eval*:

```
(define (f # $x)
    ;get-and-set the value of x
    (define x (eval $x #))
    (define y x)
    (set! z (* z 2))
    (+ x y)
    )
```

The built-in *eval* function is used to evaluate the thunk comprised of *$x* and #.

With *call-by-need*, one can define functions that exhibit short-circuiting:

```
function my-if(test,truthiness,falseness)
    {
    if (test)  ; test needed, it is evaluated
```

```
        return truthiness   ; truthiness need, falseness not needed
    else
        return falseness    ; falseness need, truthiness not needed
    }
```

Now, function calls like:

```
my-if(y == 0,result = 0,result = x / y)
```

work as intended.

Call-by-name-with-thunks

Call-by-name-with-thunks combines the semantics of *call-by-name* and *call-by-need*, the difference being that thunks are used instead of textual substitutions. in the body of the function. *Call-by-name-with-thunks* differs from *call-by-need* in that the result of evaluating a thunk is not memoized (i.e. stored for future retrieval). Instead, repeated calls for the the value of the formal parameter result in the expression in the thunk being evaluated again and again. Differences between *call-by-need* and *call-by-name-with-thunks* arise when the thunk's expression causes a change of state.

```
function f(x)
    {
    var y = x;      //x needed! Evaluate the arg in the calling env
    z = z * 2;
    return x + y;   //x needed! Evaluate the arg in the calling env
    }

var z = 5;
f(z + 3);
```

Like *call-by-name*, this function call also returns 21. One can simulate *call-by-name-with-thunks* in Scam easily:

```
(define (f # $x)
    (define y (eval $x #))
    (set! z (* z 2))
    (+ (eval $x #) y)
    )
```

Here, any time the value of the delayed argument is needed, a fresh evaluation is invoked.