Department of Computer Science and Engineering
University of California at San Diego CSE 230
Winter 1998

Problem Set 5 Solutions

1. It has been claimed that in Pascal a loop in the main program of the form **while** $\mathcal{E}$ **do** $\mathcal{C}$ is identical in effect to the procedure invocation $whiledo(\mathcal{E}, C)$ given the top-level procedure definitions

   procedure C;
   begin
   $\quad\mathcal{C}$
   end;

   procedure $whiledo(E : boolean;$ **procedure** $C)$;
   begin
   $\quad$**if** E **then**
   $\quad\quad$**begin**
   $\quad\quad\quad C; whiledo(E, C)$
   $\quad\quad$**end**
   end;

   Note that in this program text, calligraphic letters such as $\mathcal{E}$ and $\mathcal{C}$ indicate arbitrary well-formed expressions or commands, while italic letters such as E are identifiers.

   Why is this claim incorrect? Suggest a change to the semantics of Pascal that would make the claim correct. How could the program fragment be modified to make the claim correct, mimicking in standard Pascal your proposed change in semantics?

   [Solution due to Jeff McKinstry.] The problem is that the initial call to the procedure $whiledo$ causes the expression $\mathcal{E}$ to be evaluated once, in the caller's environment. The boolean result is then passed to the procedure. In order for the while loop to terminate, the expression must be reevaluated each time through the loop, but this implementation uses the initial value on every trip through the loop. The procedure $whiledo$ will either terminate immediately, if $\mathcal{E}$ initially evaluates to *false*, or it will recurse infinitely, if $\mathcal{E}$ initially evaluates to *true*.

   If Pascal implemented parameter passing by name, then the claim would be correct. Call-by-name would take the actual argument passed in for the formal parameter E, and substitute the text of the actual for E in the **if** command. The actual argument $\mathcal{E}$ would be evaluated repeatedly in the caller's environment, producing the desired results. There is no possibility of capture here because there are no local variables in the procedure $whiledo$.

   Call-by-name can be simulated by passing a function which evaluates the expression $\mathcal{E}$ in the caller's environment. The following Pascal code implements the functionality of call-by-name. This code compiles and runs, and has the desired result: it prints the letter `C` four times.

```
program main;
    var x : integer;

    procedure C;
    begin
        x := x + 1;
        writeln('C');
    end;

    function E;
    begin
        E := x < 5;
    end;

    procedure whiledo(function E : boolean; procedure C);
    begin
        if E then
            begin
                C; whiledo(E, C)
            end
    end;

begin
    x := 1;
    whiledo(E, C);
end.
```

2. Here is a Pascal procedure that is intended to swap the values of its two arguments, which are passed by reference.

```
procedure swap (var x, y : integer);
begin
        x := x + y;   y := x − y;   x := x − y
end
```

Give examples of two contexts where, for different reasons, this procedure does not work. Explain features of the Ada language which could be used to let a similar Ada procedure be correct in at least one of these contexts. (For information on Ada, see [Kamin, 1990] or [Sethi, 1989].)

[Solution due to Leesa Hicks.] The procedure *swap* will not work properly when (1) the arithmetic operations on the values of x and y exceed the capacity of *integer* variables, or (2) when the actual parameters given for x and y represent the same location, that is *swap* is invoked with the same variable for both parameters.

In the first case, $x+y$ can be too large a value to store in an integer or $x-y$ can be too small. Ada can be used to detect when an arithmetic error occurs using the *NUMERIC_ERROR* exception, but there is no means of correcting the procedure's behavior other than rewriting it to avoid using arithmetic.

In the second case, the code is executed effectively as

$$x := x + x; \;\; x := x - x; \;\; x := x - x$$

so x and y will contain 0 upon procedure exit.

To correct this in Ada, the parameters x and y can be declared as **inout**, to use call-by-value-result. Parameters defined **inout** are copied at the start of the procedure call, providing the actual parameters with distinct storage locations, enabling correct behavior of the $swap$ procedure. Upon procedure exit, the **inout** parameters are copied back to their original locations. With this change in parameter mode, the (single) original variable will contain its original value as expected, no matter what order in which the copying of arguments back is performed.

[Additional note.] Several people noted that using "exclusive or", the values of two bit vectors can be swapped without using any extra intermediate storage:

procedure $swap$ (**var** $x, y : bitvector$);
begin
    $x := x$ **xor** y; $y := x$ **xor** y; $x := x$ **xor** y
end

Exercise: can this idea be implemented in a *swap* function in C, Pascal, and/or Ada that works for all argument types (i.e. is polymorphic) and that uses no extra storage?

3. (a) According to Prashant Jain and Douglas C. Schmidt in their article *Experiences Converting a C++ Communication Software Framework to Java* in the magazine *C++ Report*, January 1997:

> In contrast to other OO languages like C++ and Ada95, Java doesn't support user-specified parameter passing modes. Instead, Java has reference semantics for object parameters, but passes these references by value. For example, in the following code, even after the `someMethod` invocation, `bar` still points to the `String` object associated with the literal string `"original"`.

```
void someMethod (String foo) {
  foo = "changed";
}

void caller () {
  String bar = "original";
  someMethod (bar);
}
```

Invent and explain a systematic way of writing methods like `SomeMethod` differently so that they can update their arguments.

[Solution due to Prashant Jain and Douglas C. Schmidt, from `http://siesta.cs.wustl.edu/~pjain/java/java_notes.html`.] We identified the following four solutions:

1. Add an extra level of indirection – We applied this common pattern and added Java wrappers around classes we wanted to pass by reference. Then, we passed instances of the wrapper class. Thus, in the example above, we created a wrapper string class called `CString`. We then passed a reference to this wrapper object that contained the actual `String`. Here's what the `CString` class looks like:

   ```
   public class CString
   {
     // Constructor.
     public CString (String s) { s_ = s; }

     // Set the underlying string.
     public void rep (String s) { s_ = s; }

     // Get the underlying string.
     public String toString () { return s_; }

     // Note, other type-specific constructors.
     // can be added.

     private String s_;
   }
   ```

   With this change, the `String` object can now be changed both by the caller and the callee. Thus, the first `someMethod` example can now be rewritten as follows:

   ```
   void someMethod (CString foo) {
     foo.rep ("changed");
   }

   void caller () {
     CString bar = new CString ("original");
     someMethod (bar);
   }
   ```

2. Use one-element array – The second solution passed the `String` object in a one-element array. Using this solution, the first `someMethod` example can be rewritten as follows:

   ```
   void someMethod (String[] foo) {
     foo[0] = "changed";
   }

   void caller () {
     String[] bar = new String[1];
   ```

```
    bar [0] = "original";
    someMethod (bar);
  }
```

Although this syntax is somewhat ugly, it is relatively concise. Moreover, we can generalize this approach to pass in multiple objects by reference via a multiple-element array.

3. Use mutator operators – Our third solution used mutator methods. For instance, in our example above, we could use `StringBuffer` instead of `String` to pass the messages between the caller and the callee. Using this solution, the above example can be rewritten as follows:

```
void someMethod (StringBuffer foo) {
  foo.setLength (0);
  foo.append ("changed");
}

void caller () {
  StringBuffer bar =
    new StringBuffer ("original");
  someMethod (bar);
}
```

   Unfortunately, the numeric wrappers (e.g., class `Double` and `Integer`) lack mutator operations. Therefore, they aren't useful for returning scalar parameters by reference (despite the claims made by certain Java books). This forces programmers to write many additional (and incompatible) wrapper classes that allow these types to be passed by reference.

4. Return objects by value – The fourth solution relies on the callee returning a modified version of the object as the return value to the caller. The caller then assigns the return value to the original object in order to "update" it. Using this solution, the above example can be rewritten as follows:

```
String someMethod (String foo) {
  return new String ("changed");
}

void caller () {
  String bar = new String ("original");

  // Call someMethod and reassign
  // bar to the return value.
  bar = someMethod (bar);
}
```

   However, to generalize this approach to return multiple values by reference requires defining even more helper classes.

(b) (This problem is due to Paul Phillips.) Another way of describing parameter passing in Java is to say that all objects are passed by reference, but references are passed by value. There is no way to pass references themselves by reference, as in C with `int * a, * b, swap(&a, &b).`

Consider the Java procedure

```
public static void swap(Object a, Object b) {
    Object temp;

    temp = a;
    a = b;
    b = temp;
}
```

and suppose that `x` and `y` are arbitrary objects. Exactly what will happen when `swap(x,y)` is called?

Is it possible to write a generic `swap` procedure in Java?

[Solution omitted.] The difficulty here is to write a `swap` procedure that is polymorphic but type-safe, i.e. a procedure that will exchange the contents of any two objects belonging to the same class, but that will refuse to exchange objects belonging to different classes.