Sample Solution to PLC Assignment #9

Andreas Borchert

May 16, 2002

Question 1
The question was what is printed by following program under the assumption of call-by-value, call-by-reference, or call-by-value-return?

(define (parameter-demo)
  (define a 1)
  (define (f b)
    (set! b (+ b 1))
    (display "a = ")(display a)
    (display ", b = ")(display b)
    (newline)
  )
  (f a)
)
(parameter-demo)

The question for call-by-value is easily answered by Scheme itself:

> (load "p.scm")
a = 1, b = 2
>

In case of call-by-reference, the inner variable b is just another name for the outer a. Hence, (set! b (+ b 1)) increments a to 2 and we get a printout of

a = 2, b = 2
However, in case of call-by-value-return, the function \( f \) works on a local copy of \( a \) and updates \( a \) only on return. Hence, the output is the same as call-by-value but \( a \) becomes 2 as soon as \( f \) returns.

**Question 2**

That was the original question:

In languages supporting call-by-reference, or call-by-value-return, a swap operation is easily written. Here is an example for C++:

```cpp
template<class T>
void swap(T& a, T& b) {
    T tmp; tmp = a; a = b; b = tmp;
}
```

Such an operation, however, becomes far more challenging with call-by-name even if it still can be done. Check out the VSi/PL language by Prof. Schreiner which, among others, supports call-by-name on request:

```
holly$ java -Dcall=Name -jar pl.jar <jensen.vpl
55
110
385
holly$
```

Write a `swap` operation for VSi/PL that works if this language operates with call-by-name.

Unfortunately, I forgot to address the issue of possible side effects in functions that are called within index expressions. This was the reason why I later on removed this question.

This problem can be nicely solved once we are sure that we can live without side effects. But let us start first with understanding why it is a problem. If we use
\texttt{swap} = \texttt{fun}(x, y, \texttt{tmp}) \{ \\
\hspace{1.1em} \texttt{tmp} = x; \\
\hspace{1.1em} x = y; \\
\hspace{1.1em} y = \texttt{tmp}; \\
\}

then we get for an invocation of \texttt{swap}(i, a[i]) following sequence of instructions:

\texttt{tmp} = i; \\
i = a[i]; \\
a[i] = \texttt{tmp}; \# \texttt{danger: i might have been changed!}

The next defense line is the introduction of more temporary variables to remember the original values of both, \(x\) and \(y\):

\texttt{swap} = \texttt{fun}(x, y, \texttt{tmpx}, \texttt{tmpy}) \{ \\
\hspace{1.1em} \texttt{tmpx} = x; \\
\hspace{1.1em} \texttt{tmpy} = y; \\
\hspace{1.1em} x = \texttt{tmpy}; \\
\hspace{1.1em} y = \texttt{tmpx}; \\
\}

But even this fails for \texttt{swap}(i, a[i]) as we still have \(y\) on the left-hand-side of the final assignment which is influenced by setting \(x\) before. However, if we are not concerned about indexing out-of-range (not an issue in VSi/PL) we can check if \(y\) is influenced by modifications of \(x\):

\texttt{swap} = \texttt{fun}(x, y, \texttt{tmpx}, \texttt{tmpy}) \{ \\
\hspace{1.1em} \texttt{tmpx} = x; \\
\hspace{1.1em} \texttt{tmpy} = y; \\
\hspace{1.1em} x = \texttt{tmpy}; \\
\hspace{1.1em} \textbf{if} (y == \texttt{tmpy}) \{ \\
\hspace{2em} \# \texttt{y} was not harmed by the modification of \texttt{x} \\
\hspace{2em} y = \texttt{tmpx}; \\
\hspace{1.1em} \} \textbf{else} \{ \\
\hspace{2em} \# \texttt{x} needs to be restored to its original value \\
\hspace{2em} x = \texttt{tmpx}; \\
\hspace{2em} \# \texttt{now we can set y} \\
\}
This solution (which, in an equivalent form, was submitted by Mao-Jen Hsu) assumes that either $x$ can influence $y$ or $y$ the variable $x$ but not both each other if we exclude the special case that $x$ and $y$ are aliases of the same variable. This assumption seems to be correct but I haven’t seen a formal proof for it yet.

But it is possible to avoid this assumption by taking the solution that was developed during the IFIP conference 1971 and presented later in the paper by A. C. Fleck, “On the Impossibility of Content Exchange Through the By-Name Parameter Transmission Mechanism”, SIGPLAN Notices, vol 11, no 11, November 1976, pp 38-41:

```plaintext
swap = fun(x, y) {
    swap2 = fun(y, x, tmp) {
        tmp = x;
        x = y;
        return tmp;
    }
    y = swap2(y, x);
}
```

The nice point of this solution is that it proceeds in following steps:

1. Evaluate $y$ in the assignment of `swap` but do not modify it yet.
2. Invoke `swap2` which sets $x$ and returns the old value of $x$.
3. Finish the assignment, i.e. assign the old value of $x$ to the old address of $y$.

This solution also avoids any out-of-range accesses and it minimizes the number of assignments. However, even this very elegant solution can be exploited by functions with side effects in index expressions as shown by the little paper of A. C. Fleck:
i = 1;
p = fun () {
    i = i + 1;
    return i;
}

swap(i, a[p()]); # or swap(a[p()], i);

Question 3
In a purely functional language without the option to assign new values to variables, there is no difference between call-by-name and normal-order evaluation. This is simply due to the fact that functions always return the same result for the same input parameters.

Both, call-by-name and delay/force delay the evaluation of a parameter to a later time when it is needed. However, in languages with side effects (such as Algol-60 or Scheme) it matters how often a parameter with possible side effects is evaluated. The delay/force mechanism of Scheme guarantees that a parameter is evaluated only once. Afterwards, the computed value is cached and returned on each subsequent use of force. The opposite is true for call-by-name: Each use causes a new evaluation. Exactly this made the previous (now retracted) question so challenging.

But the feature that the parameter is evaluated each time again is exactly the technique which made Jensen’s device possible. By modifying a variable that is referenced in another expression, we could obtain new values for the other parameter—it was evaluated over and over again without caching the result. Using delay and force, it is impossible to implement Jensen’s device in Scheme. You have to use an explicit lambda construct for this as shown in the lecture on slide 192.

The Y-combinator, as shown on slides 73 and 74, requires normal-order evaluation. You can easily translate the Lambda calculus form of the Y-combinator to Scheme:

\[ Y = \lambda y. (\lambda x. (y x) x) \lambda x. (y x) x \]

(define Y
  (lambda (y)
    ((lambda (x) (y (x x)))
     (lambda (x) (y (x x)))
     5]
(define F
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
        ; else
        (* n (f (- n 1)))
      ))
  ))

If you try this, the Scheme interpreter will immediately go into an endless loop if you attempt to apply Y to F:

(define fact (Y F)) ; infinite loop

If you were unsure about the inner workings of Y and F, you could have attempted to convert all applicative-order (i.e. call-by-value) parameter passings to normal-order (i.e. call-by-need) in a simple brute-force attempt:

; Y = Ly. (Lx.(y)(x)x) Lx.(y)(x)x
(define Y
  (lambda (y)
    ((lambda (x) ((force y)
                 (delay ((force x) (delay (force x))))))
     (delay ((lambda (x) ((force y)
                          (delay ((force x) (delay (force x)))))))
    ))
  )

(define F
  (lambda (f)
(lambda (n)
  (if (= n 0)
    1
    ; else
    (* n ((force f) (- n 1)))
  )
)
)

(define fact (Y (delay F)))

This solution would have been accepted, even if it has more delay/force pairs than necessary. To reduce the number of delay/force combinations we can start with the most obvious (delay (force x)) combinations:

; Y = Ly. (Lx.(y)(x)x) Lx.(y)(x)x
(define Y
  (lambda (y)
    ((lambda (x) ((force y)
      (delay ((force x) x)))
      (delay (lambda (x) ((force y)
        (delay ((force x) x))))))
  )
)

Next we can get rid of (force y) as the evaluation of the function passed to us poses no threat:

; Y = Ly. (Lx.(y)(x)x) Lx.(y)(x)x
(define Y
  (lambda (y)
    ((lambda (x) (y (delay ((force x) x))))
      (delay (lambda (x) (y (delay ((force x) x))))))
  )
)

This also allows us to get rid of delay within the application of the Y-combinator:
(define fact (Y F))

There are more useless delay/force combinations left within the definition of Y. In fact, we need just those delays that are paired with the force within the recursive function:

(define Y
  (lambda (y)
    (lambda (x) (y (delay (x x))))
    (lambda (x) (y (delay (x x))))
  )
)

Next, you might wonder if it is possible to get rid of the force within the recursive function. This issue can be addressed by wrapping the function parameter f in another function that applies (force f) as soon as it is needed. This wrapper function is named ff. As it is required at two places, I have wrapped up the application of ff in invoke-y:

; Y = Ly. (Lx.(y)(x)x) Lx.(y)(x)x
(define Y
  (lambda (y)
    (define (invoke-y f)
      (define (ff n) ((force f) n))
      (y ff)
    )
    (lambda (x) (invoke-y (delay (x x))))
    (lambda (x) (invoke-y (delay (x x))))
  )
)

(define F
  (lambda (f)
    (lambda (n)
      (if (= n 0)
        1
        ...
      )
    )
  )
)
(define fact (Y F))

Now, F is free from (force f). However, Y now depends on the knowledge that f takes exactly one parameter n. The original Y combinator has no such dependency.

Yes, it is possible to get completely rid of delay/force. This is not too difficult if we perform following transformations:

(force f) → (f)
(delay ...) → (lambda () ...)

Note that this is not truly equivalent as force guarantees us that f is evaluated only once. After the transformation, we get multiple evaluations. This is not a problem in this example but in the general case. Here is the result:

(define Y
  (lambda (y)
    (define (invoke-y f)
      (define (ff n) ((f) n))
      (y ff)
    )
    ((lambda (x) (invoke-y (lambda () (x x))))
      (lambda (x) (invoke-y (lambda () (x x))))
    )
  )
)

This is not the most elegant solution of a applicative-order equivalent to the Y-combinator. The paper “The Why of Y” by Richard P. Gabriel which is available at http://www.dreamsongs.com/NewFiles/WhyOfY.pdf derives following solution:
(define Y
  (lambda (f)
    (let (
      (g (lambda (h) (lambda (x) ((f (h h)) x))))
      (g g)
    )))

Again, this solution has the dependency that f has exactly one parameter (called x in this solution).

Yet another variant is presented (and derived) on http://www.ece.uc.edu/~franco/C511/html/Scheme/ycomb.html by John Franco:

(define Y
  (lambda (X)
    ((lambda (procedure)
        (X (lambda (arg) ((procedure procedure) arg)))
      (lambda (procedure)
        (X (lambda (arg) ((procedure procedure) arg)))))))