Sample Solution to PLC Assignment #5

Andreas Borchert

April 14, 2002

Question 1

As described by the hints for this question, we have to develop something of the following form:

$$\mathcal{E}[E_1 + E_2] = \lambda \sigma. \langle \text{the value, the new symbol table} \rangle$$  \hspace{1cm} (1)

Let us start with the value and here with the first expression $E_1$. If we pass our beginning state $\sigma$ to $\mathcal{E}[E_1]$ we express it in the following form:

$$\mathcal{E}[E_1] \sigma$$  \hspace{1cm} (2)

As this returns a tuple $\langle \eta_1, \sigma_1 \rangle$, we can tell that explicitly:

$$\langle \eta_1, \sigma_1 \rangle = \mathcal{E}[E_1] \sigma$$  \hspace{1cm} (3)

Given the new state $\sigma_1$ which has been changed from $\sigma$ if and only if $E_1$ had side effects, we can compute the expression $E_2$:

$$\langle \eta_2, \sigma_2 \rangle = \mathcal{E}[E_2] \sigma_1$$  \hspace{1cm} (4)

Note how the possibly modified state of $\mathcal{E}[E_1] \sigma$ is passed on to the evaluation of $E_2$. This means that we evaluate the left operand first and that the possible side effects of that evaluation are to be seen during the evaluation of the right term. Now we can start to compose everything. If we use the values above, we get following:

$$\mathcal{E}[E_1 + E_2] = \lambda \sigma. \langle \eta_1 \mid N + \eta_2 \mid N, \sigma_2 \rangle$$  \hspace{1cm} (5)

Note that at times $\mid N$ is omitted to keep the formulas short. This is pretty close to the Java-like notation introduced in the hints to this assignment:
public static TupleES E(Expression e1, Expression e2, 
SymbolTable s) {
    TupleES tuple1 = E(e1, s); // (3)
    TupleES tuple2 = E(e2, tuple1.getSymbolTable()); // (4)
    return new TupleES(
        tuple1.getValue().getIntVal() +
        tuple2.getValue().getIntVal(),
        tuple2.getSymbolTable()); // (5)
}

If we want to expand everything in one formula (you did not have to), we
start with the results of the evaluation of $E_1$:

\[
\eta_1 = \mathcal{E}[E_1] \sigma \downarrow 1 \quad (6) \\
\sigma_1 = \mathcal{E}[E_1] \sigma \downarrow 2 \quad (7)
\]

Likewise, we can define the results of the evaluation of $E_2$:

\[
\eta_2 = \mathcal{E}[E_2] \sigma_1 \downarrow 1 \quad (8) \\
\sigma_2 = \mathcal{E}[E_2] \sigma_1 \downarrow 2 \quad (9)
\]

These equations can be expanded by using (7):

\[
\eta_2 = \mathcal{E}[E_2](\mathcal{E}[E_1] \sigma_2 \downarrow 2) \downarrow 1 \quad (10) \\
\sigma_2 = \mathcal{E}[E_2](\mathcal{E}[E_1] \sigma_2 \downarrow 2) \downarrow 2 \quad (11)
\]

Hence, the addition can be written as follows, using (6) and (10):

\[
\eta_1 \upharpoonright N + \eta_2 \upharpoonright N = (\mathcal{E}[E_1] \sigma \downarrow 1) \upharpoonright N + \mathcal{E}[E_2](\mathcal{E}[E_1] \sigma \downarrow 2) \downarrow 1 \upharpoonright N \quad (12)
\]

Now we can put everything together, using (5), (12), and (11):

\[
\mathcal{E}[E_1 + E_2] = \lambda \sigma. \langle \eta_1 \upharpoonright N + \eta_2 \upharpoonright N, \sigma_2 \rangle
\]

\[
= \lambda \sigma. \langle \mathcal{E}[E_1] \sigma \downarrow 1 \upharpoonright N + (\mathcal{E}[E_2](\mathcal{E}[E_1] \sigma \downarrow 2) \downarrow 1) \upharpoonright N, \\
\mathcal{E}[E_2](\mathcal{E}[E_1] \sigma \downarrow 2) \downarrow 2 \rangle \quad (13)
\]
You are curious about other semantic equations of this extended language? Here it goes:

\[ E[0] = \lambda \sigma. (0 \text{ in } E, \sigma) \quad (14) \]
\[ E[I] = \lambda \sigma. (\sigma(I), \sigma) \quad (15) \]
\[ E[I := E] = \lambda \sigma. (E[I := E(\sigma \downarrow 1), \sigma[E[I := E] \downarrow 1/I])] \quad (16) \]
\[ S[I := E] = \lambda \gamma. ((E[I := E](\gamma \downarrow 1)) \downarrow 2, \gamma \downarrow 2, \gamma \downarrow 3) \quad (17) \]

**Question 2**

As with the design of a class in Java, the next issue, after having an interface, is the representation of state. We need four things:

- the current position \((x, y)\) which can be either represented by a small list, a small vector, or two distinct variables \(x\) and \(y\),
- the current orientation \(\Delta\) which can be represented in form of \((\Delta x, \Delta y)\), or, as just four directions need to be supported in form of a value out of \textit{north}, \textit{east}, \textit{south}, or \textit{west} (similar to enumeration types),
- a stack of positions and directions for implementing \textit{push} and \textit{pop}, and
- the minima and maxima of \(x\) and \(y\).

While implementing the directions as north, east, south, or west might sound appealing, it would end up with three case constructs, for \textit{forward}, \textit{left}, and \textit{right}. A pair \((dx, dy)\) makes \textit{forward} easy but \textit{left} and \textit{right} can no longer be expressed using a case as lists are considered as equivalent to lists only if they share the same location. So, it is appealing to solve it in the general way in form of a matrix multiplication.

In general, all linear geometric transformations, i.e. rotations, scalings, and translations, can be expressed in the form of

\[ \vec{x}' = A\vec{x} + \vec{t} \quad (18) \]

where the matrix \(A\) describes rotation and scale while \(t\) implements translation. As we are just interested in rotations of our direction vector \(\Delta\), we can simplify (18) to
\[
\begin{pmatrix}
\Delta x' \\
\Delta y'
\end{pmatrix} =
\begin{pmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{pmatrix}
\begin{pmatrix}
\Delta x \\
\Delta y
\end{pmatrix}
\]

which, for a given rotation angle \( \theta \), becomes
\[
\begin{pmatrix}
\Delta x' \\
\Delta y'
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{pmatrix}
\begin{pmatrix}
\Delta x \\
\Delta y
\end{pmatrix}
\]

A left rotation works with \( \theta = 90^\circ = \frac{\pi}{2} \):
\[
L := \begin{pmatrix}
0 & -1 \\
1 & 0
\end{pmatrix}
\]

and right rotation with \( \theta = 270^\circ = \frac{3\pi}{2} \):
\[
R := \begin{pmatrix}
0 & 1 \\
-1 & 0
\end{pmatrix}
\]

This can be expressed in Scheme, using vectors:

```scheme
(define left-rotation #( #( 0 -1) #( 1 0)))
(define right-rotation #( #( 0 1) #(-1 0)))
```

#(1 2 3) is a vector consisting of elements 1, 2, and 3. They look like lists but they do not consist of pairs but of one structure that allows random-access, using `vector-ref`. This allows to write a rotate operator:

```scheme
(define (rotate matrix delta)
  (vector
    (+
      (* (vector-ref (vector-ref matrix 0) 0) (vector-ref delta 0))
      (* (vector-ref (vector-ref matrix 0) 1) (vector-ref delta 1)))
    (+
      (* (vector-ref (vector-ref matrix 1) 0) (vector-ref delta 0))
      (* (vector-ref (vector-ref matrix 1) 1) (vector-ref delta 1))))
)
```

4
However, lists are in many contexts far more convenient than vectors, in particular if we have very small vectors only:

```scheme
(define left-rotation '(( 0 -1) ( 1 0)))
(define right-rotation '(( 0 1) (-1 0)))
(define (rotate matrix delta)
  (list
    (+
     (* (caar matrix) (car delta))
     (* (cadar matrix) (cadr delta))
    )
    (+
     (* (caadr matrix) (car delta))
     (* (cadadr matrix) (cadr delta))
    )
  )
)
```

The shortcuts for sequences of `car` and `cdr` help here, `caar` selects is equivalent to `[0][0]` in Java-notation, `cadar` equivalent to `[0][1]`, `caadr` to `[1][0]`, and `cadadr` to `[1][1]`. This allows to express a left-rotation elegantly:

```scheme
(lambda (delta) (rotate left-rotation delta))
```

A forward operation can be elegantly implemented using the `map` operator:

```scheme
(lambda (position delta) (map + position delta))
```

`map` takes an operator and one or more lists. It generates a list, where the `i`th element is computed by passing all the `i`th elements of the lists to the operator. For `position`, and `delta` being lists of length two this is equivalent to:

```scheme
(lambda (position delta)
  (list
    (+ (car position) (car delta))
    (+ (cadr position) (cadr delta))
  )
)
```
You see, *map* is ideal for vector operations like addition or subtraction and cannot be expressed that elegantly for native vectors as there is no *vector-map* provided. Of course, it would be possible to write one or to convert vectors to maps and vice versa but this simply does not pay off here.

After having seen the elegant *map* operator, you might wonder if it is possible to simplify the matrix multiplication that is used for the rotation as well. Yes, it is. Let us start with a scalar product of vectors which is part of a matrix multiplication. In general, the scalar product of two vectors \( \vec{x}, \vec{y} \) is defined to be:

\[
\vec{x}^T \vec{y} = \begin{pmatrix} x_1 & \cdots & x_n \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \sum_{i=1}^{n} x_i y_i \tag{23}
\]

We multiply also \( x_i \) with \( y_i \) for all elements of \( x \) and \( y \). This can be expressed using *map*:

\[
\text{(map } * x \ y) \tag{23}
\]

This returns a list of products which, using the + operator, can be summed up in one rush as it takes any number of arguments:

\[
\text{(apply } + \text{ (map } * x \ y)) \tag{23}
\]

Note that + expects individual arguments but not a list with all operands. *apply* solves this problem by taking a list and converting this into individual arguments that are fed to the operator.

If we multiply a matrix with a vector, we get a vector which consists of the scalar products of the first row of the matrix with the vector, the second row of the matrix with the vector etc. This simply looks like another application of *map* which gets the above scalar product as operator and two lists, a list of rows (i.e. the matrix), and a list of the replicated vector (as this vector needs to be multiplied with each of the rows). The only tricky point left is the replication of the vector. This job can be done by

\[
\text{(map } (\lambda \text{ (element) vector) vector)} \tag{23}
\]

where each element of the vector is replaced by the entire vector, giving a matrix. Putting everything together:
The rotation operations can be simplified then as follows, making *rotate* superfluous:

```scheme
(define (left) (set! delta (mult-matrix-vector left-rotation delta)))
(define (right) (set! delta (mult-matrix-vector right-rotation delta)))
```

This allows us to formulate a simple turtle that supports just movements and rotations:

```scheme
(define (create-simple-turtle position delta)
  (define (forward) (set! position (map + position delta)))
  (define (left) (set! delta (mult-matrix-vector left-rotation delta)))
  (define (right) (set! delta (mult-matrix-vector right-rotation delta)))
  (lambda (operation)
    (case operation
      ((forward) (forward))
      ((left) (left))
      ((right) (right))
      ((x) (car position))
      ((y) (cadr position))
      ((xy) position)
      ((dx) (car delta))
      ((dy) (cadr delta))
      ((clone) (create-simple-turtle position delta))
    ))
)
```

This allows us to set up some initial state:
\begin{verbatim}
(define min-xy '(0 0))
(define max-xy '(0 0))
(define stack (create-simple-turtle '(0 0) '(0 -1)))
(define (turtle) (car stack))

With \textit{stack} we have a stack of simple turtles, the top-most is returned by
the procedure \textit{turtle}. Just \textit{forward} needs to be augmented to keep track of
the minimal and maximal coordinates:

\begin{verbatim}
(define (forward)
  ((car stack) 'forward)
  (set! min-xy (map min min-xy ((turtle) 'xy)))
  (set! max-xy (map max max-xy ((turtle) 'xy)))
)
\end{verbatim}

The operators \textit{min} and \textit{max} accept a list of numbers of which they return
the minimal and maximal value, respectively. They can be easily combined
with \textit{map} to be applied on vectors where the \textit{i}th resulting element is the
minimum of the \textit{i}th elements of the input vectors.

Finally, we can formulate the \textit{case} construct that maps all the method
names to the associated operations:

\begin{verbatim}
(lambda (operation)
  (case operation
    ((push) (set! stack (cons ((turtle) 'clone) stack)))
    ((pop) (set! stack (cdr stack)))
    ((forward) (forward))
    ((left) ((turtle) 'left))
    ((right) ((turtle) 'right))
    ((x) ((turtle) 'x))
    ((y) ((turtle) 'y))
    ((minx) (car min-xy))
    ((maxx) (car max-xy))
    ((miny) (cadr min-xy))
    ((maxy) (cadr max-xy))
  )
)
\end{verbatim}
\end{verbatim}