SICP section 3.3.5

October 18, 2007 at 06:18 Tags SICP

Similarly to the previous section, I'm going to implement the objects usingCLOS classes and generic methods.

First, the connector:

```
(defclass connector ()
  ( (value
      :initform nil
      :accessor value
      :documentation
        "Current value of the connector")
    (informant
      :initform nil
      :accessor informant
      :documentation
        "The object that set the connector's value")
    (constraints
      :initform '()
      :accessor constraints
      :documentation
        "The constraints in which the connector
        participages")))

(defun make-connector ()
  (make-instance 'connector))

(defmethod has-value? ((c connector))
  (if (informant c)
    t
    '()))

(defmethod set-value! ((c connector) newval setter)
  (cond ((not (has-value? c))
          (setf (value c) newval)
          (setf (informant c) setter)
          (for-each-except
            setter
            #'process-new-value
            (constraints c)))
        ((not (= (value c) newval))
        (error "Contradiction"
          (list (value c) newval)))))

(defmethod forget-value! ((c connector) retractor)
  (when (eq retractor (informant c))
    (setf (informant c) nil)
    (for-each-except
      retractor
      #'process-forget-value
      (constraints c))))

(defmethod connect ((c connector) new-constraint)
  (if (not (member new-constraint (constraints c)))
    (push new-constraint (constraints c)))
  (if (has-value? c)
    (process-new-value new-constraint)))

(defun for-each-except (exception proc seq)
  (dolist (obj seq)
    (unless (eq obj exception)
      (funcall proc obj))))

(defmacro with-connectors ((&rest names) &body body)
  `(let ,(loop for n in names collect `(,n (make-connector)))
    ,@body))
```

This is just a simple class with some methods. Note the`with-connectors` macro that will help us write complex constraint blocks with less repetitive typing. We'll see its use shortly.

As for the constraints, we'll going to use a little more ofCLOS's power by actually employing the dispatching of generic methods. First, here are the two methods our constraints must implement:

```
(defgeneric process-new-value (constraint)
  (:documentation
    "Called when one of the connectors attached to
    this constraint has a new value"))

(defgeneric process-forget-value (constraint)
  (:documentation
    "Ask the attached connectors to forget their values"))
```

Now, we can write our constraint classes and methods that implement these generics. In contrast to C++ and Java, these classes don't have to be siblings. In fact, they can be any classes at all as long as they implement these two methods. Such approach has been recently brought to fame by Ruby, and it's called *duck typing*. Here are the adder and multipler constraints, which are actually quite similar:

```
(defclass adder-constraint ()
  ( (a1
      :accessor a1
      :initarg :a1
      :documentation "First addend")
```

```
    (a2
      :accessor a2
      :initarg :a2
      :documentation "Second addend")
    (sum
      :accessor sum
      :initarg :sum
      :documentation "Sum of addends")))

(defun adder (a1 a2 sum)
  (let ((cs (make-instance
             'adder-constraint
             :a1 a1 :a2 a2 :sum sum)))
    (connect a1 cs)
    (connect a2 cs)
    (connect sum cs)
    cs))

(defmethod process-new-value ((cs adder-constraint))
  (with-accessors ((a1 a1) (a2 a2) (sum sum)) cs
    (cond ((and (has-value? a1) (has-value? a2))
            (set-value! sum
                        (+ (value a1) (value a2))
                        cs))
          ((and (has-value? a1) (has-value? sum))
            (set-value! a2
                        (- (value sum) (value a1))
                        cs))
          ((and (has-value? a2) (has-value? sum))
            (set-value! a1
                        (- (value sum) (value a2))
                        cs)))))

(defmethod process-forget-value ((cs adder-constraint))
  (forget-value! (a1 cs) cs)
  (forget-value! (a2 cs) cs)
  (forget-value! (sum cs) cs)
  (process-new-value cs))

(defclass multiplier-constraint ()
  ( (m1
      :accessor m1
      :initarg :m1
      :documentation "First multiplicand")
    (m2
      :accessor m2
      :initarg :m2
      :documentation "Second multiplicand")
    (product
      :accessor product
      :initarg :product
      :documentation "Product of multiplicands")))
```

```
(defun multiplier (m1 m2 product)
  (let ((cs (make-instance
              'multiplier-constraint
              :m1 m1 :m2 m2 :product product)))
    (connect m1 cs)
    (connect m2 cs)
    (connect product cs)
    cs))

(defmethod process-new-value ((cs multiplier-constraint))
  (with-accessors ((m1 m1) (m2 m2) (product product)) cs
    (cond ((or  (and (has-value? m1) (= (value m1) 0))
                (and (has-value? m2) (= (value m2) 0)))
            (set-value! product 0 cs))
          ((and (has-value? m1) (has-value? m2))
            (set-value! product
                        (* (value m1) (value m2)) cs))
          ((and (has-value? product) (has-value? m1))
            (set-value! m2
                        (/ (value product) (value m1) cs)))
          ((and (has-value? product) (has-value? m2))
            (set-value! m1
                        (/ (value product) (value m2)) cs)))))

(defmethod process-forget-value ((cs multiplier-constraint))
  (forget-value! (m1 cs) cs)
  (forget-value! (m2 cs) cs)
  (forget-value! (product cs) cs)
  (process-new-value cs))
```

I suppose these do could be abstracted in some way by a generic “binary operation” constraint. This will be left as an exercise to the reader.

The following two constrains are, in fact, quite different. But they still implement the two methods, so they are lawful constraints as far as we’re concerned:

```
(defclass constant-constraint () ())

(defun constant (value connector)
  (let ((cs (make-instance 'constant-constraint)))
    (connect connector cs)
    (set-value! connector value cs)
    cs))

(defmethod process-new-value ((cs constant-constraint))
  (error "Unable to set new value for CONSTANT"))

(defmethod process-forget-value ((cs constant-constraint))
  (error "Unable to forget a value for CONSTANT"))

(defclass probe-constraint ()
  ( (name :initarg :name :accessor name)
    (connector :initarg :connector :accessor connector)))

(defun probe (name connector)
  (let ((cs (make-instance
              'probe-constraint
              :name name
              :connector connector)))
    (connect connector cs)
    cs))

(defmethod print-probe ((cs probe-constraint) value)
  (format t "~%Probe: ~a = ~a" (name cs) value))

(defmethod process-new-value ((cs probe-constraint))
  (print-probe cs (value (connector cs))))

(defmethod process-forget-value ((cs probe-constraint))
  (print-probe cs "?"))
```

Here’s a sample code that demonstrates how these are used (note the usage of `with-connectors` in `celsius-fahrenheit-converter`):

```
(defun celsius-fahrenheit-converter (c f)
  (with-connectors (u v w x y)
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))

(deflex c (make-connector))
(deflex f (make-connector))
(celsius-fahrenheit-converter c f)

(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)

(set-value! C 25 'user)
(forget-value! C 'user)

(set-value! F 212 'user)
```

This prints:

```
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
Probe: Fahrenheit temp = 212
Probe: Celsius temp = 100
```

Note my usage of `deflex`. It is a macro for defining lexical variables in the global environment. I became painfully disillusioned with the functioning of `defvar` in interpreted code (run byCLISP) – its behavior contains non trivial gotchas caused by the dynamic scope. `deflex` is a useful macro I picked from comp.lang.lisp, it does the same work for me and behaves in a more expected way:

```
(defmacro deflex (var val &optional (doc nil docp))
  "Define a top level (global) lexical VAR with
  initial value VAL, which is assigned
  unconditionally as with DEFPARAMETER. If a DOC
  string is provided, it is attached to both the
  name |VAR| and the name *STORAGE-FOR-DEFLEX-VAR-|VAR|*
  as a documentation string of kind 'VARIABLE.
  The new VAR will have lexical scope and thus may be
  shadowed by LET bindings without affecting its
  dynamic (global) value."
  (let* ((s0 (symbol-name '#:*storage-for-deflex-var-))
         (s1 (symbol-name var))
         (s2 (symbol-name '#:*))
         (s3 (symbol-package var))
         (backing-var
          (intern (concatenate 'string s0 s1 s2) s3)))
    ; Note: The DEFINE-SYMBOL-MACRO must be the last
    ; thing we do so that the value of the form is the
    ; symbol VAR.
    ; (print (concatenate 'string s0 s1 s2))
    (if docp
      `(progn
         (defparameter ,backing-var ,val ,doc)
         (setf (documentation ',var 'variable) ,doc)
         (define-symbol-macro ,var ,backing-var))
      `(progn
         (defparameter ,backing-var ,val)
         (define-symbol-macro ,var ,backing-var)))))
```

Exercise 3.33

```
(defun averager (a b c)
  (with-connectors (half sum)
    (constant 0.5 half)
    (adder a b sum)
    (multiplier half sum c)
    'ok))
```

Exercise 3.34

The simplest incarnation of the flaw can be demonstrated thus:

```
(with-connectors (a b)
  (probe "a" a)
  (probe "b" b)
  (squarer a b)
  (set-value! b 64 'user))
```

This prints only:

```
Probe: b = 64
```

Why ? Look at the way `process-new-value` for `multiplier` is implemented. It expects to see values set for two of its terminals, and computes the third. But in the example above, only one of the three terminals is set. Sure, the other two terminals can be considered equivalent, but the multiplier constraint has no idea about this. It sees that only one terminal is set and doesn't compute the others, because as far as it's concerns, the data is incomplete.

Exercise 3.35

```
(defclass squarer-constraint ()
  ( (root
      :accessor root
      :initarg :root
      :documentation "The square root site")
    (sq
      :accessor sq
      :initarg :sq
      :documentation "The square side")))

(defun squarer (root sq)
  (let ((cs (make-instance
              'squarer-constraint
              :root root :sq sq)))
    (connect root cs)
    (connect sq cs)
    cs))

(defmethod process-new-value ((cs squarer-constraint))
  (with-accessors ((root root) (sq sq)) cs
    (if (has-value? sq)
      (if (< (value sq) 0)
        (error "square less than 0 -- SQUARER"
          (value sq))
        (set-value! root (sqrt (value sq)) cs))
      (when (has-value? root)
        (set-value! sq (square (value root)) cs)))))

(defmethod process-forget-value ((cs squarer-constraint))
  (forget-value! (sq cs) cs)
  (forget-value! (root cs) cs)
  (process-new-value cs))
```

Exercise 3.36

Obviously the env diagram wouldn't be too relevant with my implementation, because theCLOS objects hide all the details of closures holding data.

Exercise 3.37

```
(defun cv (val)
  (with-connectors (c)
    (constant val c)
    c))

(defun c+ (x y)
  (with-connectors (z)
    (adder x y z)
    z))

(defun c- (x y)
  (with-connectors (z)
    (adder z y x)
    z))

(defun c* (x y)
  (with-connectors (z)
    (multiplier x y z)
    z))

(defun c/ (x y)
  (with-connectors (z)
    (multiplier z y x)
    z))

(defun celsius-fahrenheit-converter2 (x)
  (c+ (c* (c/ (cv 9) (cv 5))
          x)
      (cv 32)))
```

For comments, please send me an email.

Back to top