Cells tutorial

19th August 2008

Contents

1 Introduction

1.1 What's cells?

Cells is a Common Lisp library that extends the language, and in particular its object system, to let you write dataflow-driven programs. What does this mean? This means that the flow of control of the program depends no more on the sequence of function/method calls, but on the data. Cells lets you specify the dependence beetwen different slots[1] in a family of classes. Once these constraints

[1] A slot is the Common Lisp equivalent of a class instance variable in other languages

have been registered, the cells system will take care of them, and will recalculate a value when some data on which it depends has changed. As a consequence, the programmer just has to tell the system the *relationship* between the data, the burden of maintaining them true is handled automatically by cells.

1.2 How could it improve your programs?

Cells may not be the panacea of programming, but it sure helps a lot in contexts where keeping a set of values consistent is crucial. A particular set of applications where this is important are graphical applications[2], where you need to maintain consistency between what the user sees and the real values held by the program in its internal data structures. An example is the state of the 'Cut' menu entry in an editor: it is usually clickable when the user has selected a piece of text and not clickable in all the other cases. In a normal application, to achieve this behavior you would need to track all the methods and all the user actions that could modify the region of text currently being selected, and add activate/disactivate calls in all those places to keep the menu entry in a consistent state. With cells, you just need to tell the system that the state of the menu depends on the length of the current text selection: if the length is 0 then the state is 'deactivated', else it is 'activated'. Now you can safely work on the rest of the application ignoring the state of the menu: it will be automatically recalculated every time the length of the current selection varies. Moreover, everything relating to the menu entry is placed near its definition, and not scattered across different functions/methods.

2 Installation

The installation is quite simple once you have a working Common Lisp system. Here I will assume that you've got a working copy of SBCL[3]. First of all, download cells: you can get the latest version at `http://common-lisp.net/cgi-bin/viewcvs.cgi/cells/?root=cells`. Then enter the directory ~/.sbcl/site and unpack cells:

```
$ cd ~/.sbcl/site
$ tar -zxvf ~/cells.tar.gz
```

Now be sure that ASDF will be able to find it:

```
$ cd ~/.sbcl/systems
$ for a in ‘find ~/.sbcl/site/cells/ -name “*.asdf”‘ \

    do ln -sf $a . \
done
```

[2] See the cells-gtk project: `http://common-lisp.net/project/cells-gtk`
[3] SBCL `http://www.sbcl.org`

After that, start SBCL and evaluate the following expressions:

```
> (require :asdf)
NIL
> (asdf:oos 'asdf:load-op :cells)
(some output will follow)
```

If everything went right cells should be up and running.

3 Our first cells program

3.1 The program

Write the following piece of code in a file named hello-cells.lisp:

```
(defmodel hello-cells ()
  ((num :accessor num :initarg :num :initform (c-in 0))
   (square-num :accessor square-num
               :initform (c? (* (num self) (num self))))))
(defun hello ()
  (let ((h (make-instance 'hello-cells)))
    (dolist (n '(10 20 30 40 50 60 60))
      (setf (num h) n)
      (format t "num is ~a and square-num is ~a~%" (num h) (square-num h)))))
```

Now start the SBCL interpreter in the same directory and evaluate the following:

```
> (asdf:oos 'asdf:load-op :cells)
...
> (use-package :cells)
T
> (load "hello-cells.lisp")
...
T
> (hello)
num is 10 and square-num is 100
num is 20 and square-num is 400
num is 30 and square-num is 900
num is 40 and square-num is 1600
num is 50 and square-num is 2500
num is 60 and square-num is 3600
num is 60 and square-num is 3600
NIL
```

What happens within the function 'hello'? First, an object of type hello-cells is created. After that the program iterates over the contents of the list '(10 20 30 40 50 60 60), and every number is used to set the num slot of the object h. Then the num slot is printed together with the slot square-num. The printed value of the slot num gives us no surprise: it has the value we gave it. This doesn't hold for the slot square-num, though: we never gave it a value within the loop, but it always holds the square of the slot num! This is just cells working for us: we told the system that the relation

$$num * num = squarenum$$

must hold, and every time num changes, the expression (* (num self) (num self)) is re-evaluated. Note that the relation isn't a mathematical equation: you can't change square-num and expect to find its square root in num.

3.2 The program line-by-line

Lets now analyze the program. The very first line uses the construct defmodel:

```
(defmodel hello-cells ()
```

defmodel is very similar to defclass and everything valid in a defclass construct is valid within defmodel[4]. The main difference is that all the slots defined within it will be tracked by cells, except slots that are explicitly declared to be ignored by the system by specifying :cell nil in the definition.

```
((num :accessor num :initarg :num :initform (c-in 0))
```

Here we define the slot num as we would do within a standard class declaration. The difference is in its initialization expression: instead of the number 0 we have (c-in 0). Why? (c-in <expr>) is a construct that tells cells that the value of num may be changed, so whenever it does change a re-evaluation of all the slots that depend on it must be triggered. If we did just write 0 instead of (c-in 0) a runtime error would have been raised during the execution of (setf (num h) ...). So, when a slot is writable it must be signalled to cells with the (c-in ...) construct. This is necessary to let cells do some optimizations like avoiding to remember dependencies on slots that will never change. Slots initialized with c-in are usually called "input cells".

```
(square-num :accessor square-num
            :initform (c? (* (num self) (num self)))))
```

Now we define the slot square-num. There are two things to note here: (c? <expr>) and 'self'.The first is a construct that says: "To calculate the value of square-num, evaluate the expression <expr>". Within (c? ...) the variable self is bound to the object itself. (c? ...) automatically tracks any dependency, in this case the dependency on the value of num: when num changes, (* (num self) (num self)) will be re-evaluated. Slots initialized with c? are called "ruled cells".

[4] defmodel is a layer built on top of defclass

```
(let ((h (make-instance 'hello-cells)))
```

Here we use the function (make-instance <model-name> *args**), to create an object of type <model-name>, in this case hello-cells, as we would do to instantiate a normal class. You could specify an initial value for num now:

```
(let ((h (make-instance 'hello-cells :num (c-in 50))))
```

Note that you *must* repeat the (c-in ...) construct. This is because the behavior of the slot (input cell, constant, ruled cell) is decided on a per instance basis, not on a per class basis. This means that, in our example, we could have two objects of type hello-cells, one where the slot num is settable and one where it is has a constant value. When an object is created, all the values of its slots are computed for the first time, in this case the expression (* (num self) (num self)) is evaluated and the value given to the slot square-num.

```
(setf (num h) n)
```

This expression sets the value of the slot num to n. This is when cells comes into action: square-num depends on num, so (* (num self) (num self)) is re-evaluated after n has changed.

```
(format t "num is ~a and square-num is ~a~%" (num h) (square-num h))
```

Finally, we print the values of the two slots and discover that the value of square-num is correctly the square of num.

As a side note, you can reset the cells system by calling (cell-reset):

```
> (cells-reset)
NIL
```

This could be necessary after an error has corrupted the system and cells doesn't seem to work correctly anymore. It's also a good practice to reset the system before running code that uses cells.

4 The family system

Objects whose type have been defined using defmodel can be organized in families. A family is a tree of model instances (*not* of model classes!) that can reference each other using the functions (fm-other ...), (fm^ ...) and others. You can specify the family tree at object creation time passing a list of children to the argument :kids. Alternatively, you can access the slot .kids (automatically created by defmodel) and set it at runtime to change the family components. .kids is, by default, a slot of type c-in, and you can access it through the method (kids object). You can change the .kids slot to be of a type other than c-in as you could do with any other slot. To access the members of a family you can give them a name with the argument :md-name and then reference them by their name. Another way to access them is through their type: you could say, for example, "give me all the successors of type my-type". To use these features your models must inherit from the model 'family'. Models that inherit from family have also a .value slot associated, which you can access through the method (value self)[5]. The following example shows some of these things in action:

```
(defmodel node (family)
  ((val :initform (c-in nil) :initarg :val)))
(defun math-op-family ()
  (let ((root
         (make-instance
          'node
          :val (c? (apply #'+ (mapcar #'val (kids self))))
          :kids
          (c?
            (the-kids
              (make-kid 'node :md-name :n5 :val (c-in 5))
              (make-kid
               'node
               :val (c? (apply #'* (mapcar #'val (kids self))))
               :kids
               (c?
                 (the-kids
                  (make-kid 'node :md-name :n7 :val (c-in 7))
                  (make-kid 'node :md-name :n9 :val (c-in 9))))))))))
    (format t "value of the tree is ~a~%" (val root))
    (setf (val (fm-other :n7 :starting root)) 10)
    (format t "new value of the tree is ~a~%" (val root))))
```

Write it in a file (in this case hello-cells.lisp) and load it:

```
> (load "hello-cells.lisp")
T
> (math-op-family)
value of the tree is 68
new value of the tree is 95
NIL
```

Lets' see the most important parts of the program:

```
(defmodel node (family)
  ((val :initform (c-in nil) :initarg :val)))
```

Here we define the model node: we plan to build a family of nodes, so we inherit from the model family. The slot val will contain the value of the node.

[5]In older releases of cells you had to use (md-value self) instead

```
(make-instance
 'node
 :val (c? (apply #'+ (mapcar #'val (kids self))))
```

Now we create the main node: its value is defined as the sum of all its children values. To get the children list we use the method (kids self).

```
:kids
(c?
  (the-kids
```

We specify the children list using the :kids argument. the-kids builds a list of children using the following arguments. the-kids also removes nil kids and if an argument is a list then it is flattened, e.g. (the-kids (list k1 (list (list k2 nil) k3))) will return a list with the kids k1, k2 and k3.

```
(make-kid 'node :md-name :n5 :val (c-in 5))
```

This is the first child of the main node: we give it a name with the :md-name argument to reference the node through it in the future. To create an instance of a model intended to be a child you must specify to make-instance its parent through the argument :fm-parent. make-kid does this for us passing self as the parent.

```
(make-kid
 'node
 :val (c? (apply #'* (mapcar #'val (kids self))))
 :kids
 (c?
   (the-kids
     (make-kid 'node :md-name :n7 :val (c-in 7))
     (make-kid 'node :md-name :n9 :val (c-in 9)))))
```

The second child of the main node has two children and its value is the product of their values.

```
(format t "value of the tree is ~a~%" (val root))
(setf (val (fm-other :n7 :starting root)) 10)
(format t "new value of the tree is ~a~%" (val root))))
```

The body of the function prints the value of the tree, and through the output you can see that it depends correctly on the values of its children. Then we change the value of the node named :n7 and see that the new output has changed accordingly. (fm-other <member-name> <starting-point>) searches the family tree starting from <starting-point>, and returns the object named <member-name>. If it is not found, and error is raised. <starting-point> is optional, and it defaults to 'self'. We used fm-other outside of a defmodel, so there is no self and we must supply a starting point.

5 Defining an observer

Cells lets you define a function to execute immediately after a c-in slot is modified. This function is called an "observer". To define it, use the defobserver construct:

```
(defobserver <slot-name> (&optional (<self> self)
                                    (<new-value> old-value)
                                    (<old-value> new-value)
                                    (<old-value-boundp> old-value-boundp))
  <function-body>)
```

This function will be executed every time the slot <slot-name> of an object of type <model-name> is modified. <old-value> will hold the previous value of the slot, <new-value> the new one and <old-value-boundp> will be nil if this is the first time the slot gets a value and t otherwise. If not given, <self>, <new-value>, <old-value> and <old-value-boundp> will default to 'self', 'new-value', 'old-value' and 'old-value-bound-p'. In older releases of cells defobserver was called def-c-output.

Suppose we want to log all the values that the num slot assumes: we can do this defining an observer function. Add the following lines to hello-cells.lisp:

```
(defobserver num ((self hello-cells))
  (format t "new value of num is: ~a~%" new-value))
```

Now reload the file and try running (hello) again:

```
> (load "hello-cells.lisp")
T
> (hello)
new value of num is: 0
new value of num is: 10
num is 10 and square-num is 100
new value of num is: 20
num is 20 and square-num is 400
new value of num is: 30
num is 30 and square-num is 900
new value of num is: 40
num is 40 and square-num is 1600
new value of num is: 50
num is 50 and square-num is 2500
new value of num is: 60
num is 60 and square-num is 3600
num is 60 and square-num is 3600
NIL
```

As you can see from the output, every time we set (num h) with a different value, the action previously defined is called. This also happens when (num h) is initialized for the first time at object creation time. You may have noted that when we set (num h) to 60 for the second time, the observer function isn't called: this is because when you set a slot to a new value that is the same (according to the function eql) as its old one, the change isn't propagated because there is no need to propagate it: it didn't change!

Now look at the following piece of code:

```
(defmodel str-model ()
  ((str :accessor str :initform (c-in "") :initarg :str)
   (rev-str :accessor rev-str :initform (c? (reverse (str self))))))
(defobserver str ()
  (format t "changed!~%"))
(defun try-str-model ()
  (let ((s (make-instance 'str-model)))
    (dolist (l `("Hello!" "Bye"
                 ,(concatenate 'string "By" "e") "!olleH"))
      (setf (str s) l)
      (format t "str is \"~a\", rev-str is \"~a\"~%"
              (str s) (rev-str s)))))
```

It does nothing new: it constrains rev-str to be the reverse of str, creates an instance of str-model and prints some strings together with their reverse. It also logs every time it needs to compute the reversed string. Note that the second and the third strings of the list are actually equal. Lets try to run the code (supposing you wrote it in hello-cells.lisp):

```
> (load "hello-cells.lisp")
T
> (try-str-model)
changed!
changed!
str is "Hello!", rev-str is "!olleH"
changed!
str is "Bye", rev-str is "eyB"
changed!
str is "Bye", rev-str is "eyB"
changed!
str is "!olleH", rev-str is "Hello!"
NIL
```

The reversed string is calculated *every* time we set (str s), even when we're changing it from "Bye" to "Bye". But "Bye" and "Bye" are equal! Why do we need to waste time reversing it twice? Because cells by default uses eql to test for equality and if two strings aren't the same string (i.e. they don't have the same memory address) eql considers them to be different. The following piece of code shows us another problem: suppose we change

```
`("Hello!" "Bye" ,(concatenate 'string "By" "e") "!olleH")
```

to

```
`("Hello!" "Bye" "Bye" "!olleH")
```

depending on the Common Lisp implementation you run the program on you'll have a different output! Solving the problem is easy, we just need to use equal instead of eql as the equality function. To supply your own equality function pass it to the :unchanged-if argument in the slot definition:

```
(str :accessor str :initform (c-in "") :initarg :str
     :unchanged-if #'equal)
```

Now we get the same expected result on any implementation:

```
changed!
changed!
str is "Hello!", rev-str is "!olleH"
changed!
str is "Bye", rev-str is "eyB"
str is "Bye", rev-str is "eyB"
changed!
str is "!olleH", rev-str is "Hello!"
NIL
```

The equality function must accept two values: the new value of the slot and the old one.

6 Lazy cells

Ruled cells are evaluated, as we have already seen, at instance creation time and after dependent cells change. However, you may want to *not* evaluate a ruled cell until it is really needed, i.e. when the program asks for its value. To achieve such a behavior, you can use lazy cells. There are three types of them, depending on their laziness:

1. :once-asked this will get evaluated/observed on initialization, but won't be reevaluated immediately if dependencies change, rather only when read by application code.

2. :until-asked this does not get evaluated/observed until read by application code, but then it becomes un-lazy, eagerly reevaluated as soon as any dependency changes (not waiting until asked).

3. :always this isn't evaluated/observed until read, and not reevaluated until read after a dependency changes.

There are two ways in which a cell can be lazy: by not being evaluated immediately after its creation and by not responding to dependencies change. In both cases, when the program asks for its value, the lazy cell is evaluated. The first type embodies only the second way, the second type only the first way and the third type is lazy in both ways. The following example shows the initialization phase lazy cells:

```
(defmodel lazy-test ()
  ((lazy-1 :accessor lazy-1 :initform (c-formula (:lazy :once-asked)
                                        (append (val self) (list '!!))))
   (lazy-2 :accessor lazy-2 :initform (c_? (val self)))
   (lazy-3 :accessor lazy-3 :initform (c?_ (reverse (val self))))
   (val :accessor val :initarg :val :initform (c-in nil))))
(defobserver lazy-1 ()
  (format t "evaluating lazy-1!~%"))
(defobserver lazy-2 ()
  (format t "evaluating lazy-2!~%"))
(defobserver lazy-3 ()
  (format t "evaluating lazy-3!~%"))
(defun print-lazies (l)
  (format t "Printing all the values:~%")
  (format t "lazy-3: ~a~%" (lazy-3 l))
  (format t "lazy-2: ~a~%" (lazy-2 l))
  (format t "lazy-1: ~a~%" (lazy-1 l)))
(defun try-lazies ()
  (let ((l (make-instance 'lazy-test :val (c-in '(Im very lazy!)))))
    (format t "Initialization finished~%")
    (print-lazies l)
    (format t "Changing val~%")
    (setf (val l) '(who will be evaluated?))
    (print-lazies l)))
```

As usual, load it and run it:

```
> (load "hello-cells.lisp")
T
> (try-lazies)
evaluating lazy-1!
Initialization finished
Printing all the values:
evaluating lazy-3!
lazy-3: (LAZY! VERY IM)
evaluating lazy-2!
lazy-2: (IM VERY LAZY!)
lazy-1: (IM VERY LAZY! !!)
Changing val
evaluating lazy-2!
```

```
Printing all the values:
evaluating lazy-3!
lazy-3: (EVALUATED? BE WILL WHO)
lazy-2: (WHO WILL BE EVALUATED?)
evaluating lazy-1!
lazy-1: (WHO WILL BE EVALUATED? !!)
NIL
```

As you can see from the code to declare a ruled cell to be lazy you just need to use the three constructs (c-formula (:lazy :one-asked) ...), (c_? ...) and (c?_ ...) for :once-asked, :until-asked and :always lazy cells, respectively. lazy-1 is evaluated immediately, lazy-2 and lazy-3 only when they are needed by format. After setting (val l), on which all the lazy cells depend, lazy-2 is re-evaluated immediately because it is of type :until-asked, while lazy-1 becomes lazy and lazy-3 remains lazy, so these two postpone evaluation until we ask for their values in the call to format.

As a side note, such short names may not be very easy to remember and to read, but those constructs are so common that you'll find yourself using them a lot, and you'll appreciate their conciseness. If you still prefer long descriptive names, though, you can use the c-formula construct instead of c?/c_?/c?_ and c-input instead of c-in (see the "Functions & macros reference" section).

7 Functions & macros reference

Here follows a quick reference of the main functions and macros.

7.1 Main

defmodel

```
(defmodel <model-name> (<superclass>*)
  (<slot-definition>*)
  <other-optional-arguments>)
```

(Macro) Defines a new model. It has the same structure and the accept the same options of a class definition. <slot-definition> accepts the special argument :cell that lets you declare what kind of slot it is. The default is a normal cell slot. Other options include:

1. :cell nil the slot will be ignored by the constraints-handling system

2. :cell :ephemeral when an ephemeral slot is changed, everything works as with a normal cell, but after the propagation has ended, its value will become nil. They are useful to model events.

There are other types of cells (delta, lazy, etc.) not covered here.

c-in

```
(c-in <expr>)
```

(Macro) Initializes a cell slot with the value expr. When a cell slot initialized with c-in changes, dependant cells will be recalculated. The value of a cell slot initialized with c-in can be setted.

c-input

```
(c-input (&rest args) &optional value)
```

(Macro) Same as c-in, but it lets specify extra arguments, and value is optional. If it is not given, the slot will be unbound and any access to it will result into an error.

c?

```
(c? <body>)
```

(Macro) Initializes a cell slot with the value of <body>. If <body> references input cell slots, it will be recalculated whenever those slots change. Within c? you have access to the variable self, representing the current object

c-formula

```
(c-formula (<options>) <body>)
```

(Macro) Same as c?, but lets you specify extra options. For example, the option :inputp lets you build a cell that behaves like a cell initialized with both c? and c-in. Another useful option is :lazy that lets you specify the laziness of the cell: nil, t, :once-asked, :until-asked or :always.

not-to-be

```
(not-to-be <object>)
```

(Function) Tells cells to stop handling <object>.

defobserver

```
(defobserver <slot-name> (&optional (<self> self)
                                    (<new-value> old-value)
                                    (<old-value> new-value)
                                    (<old-value-boundp> old-value-boundp))
  <function-body>)
```

(Macro) Defines a function that is called every time the slot <slot-name> changes. In previous versions of cells it were called def-c-output.

7.2 Family models

The following only works for models that inherit from family.

make-part

```
(make-part <md-name> <model-name> &rest <args>)
```

(Function) Creates an instance of <model-name> with :md-name set to <md-name>. <args> are passed to make-instance.

mk-part

```
(mk-part <md-name> (<model-name>) &rest <args>)
```

(Macro) Same as make-part, but sets the parent to self

the-kids

```
(the-kids &rest <kids>)
```

(Macro) Builds a list of kids. <kids> may contain objects or nested lists of objects.

make-kid

```
(make-kid <model-name> &rest <args>)
```

(Macro) The same as (make-instance <model-name> <args> :fm-parent self).

kids

```
(kids <object>)
```

(Method) Gives access to <object>'s children.

kid1,kid2,last-kid

```
(kid1 <object>)
(kid2 <object>)
(last-kid <object>
```

(Function) Gives access, respectively, to <object>'s first, second and last child

^k1,^k2,^k-last

```
(^k1)
(^k2)
(^k-last)
```

(Macro) Shortcuts for (kid1 self), (kid2 self) and (last-kid self) fm-parent

```
(fm-parent &optional (<object> self))
```

(Method) Gives access to <object>'s parent.

fm-other

```
(fm-other <name> &optional (<starting-point> self))
```

(Macro) Looks for an object named <name> within <starting-point>'s family.

fm^

```
(fm^ <name> &optional (<starting-point> self))
```

(Macro) Same as (fm-other <name> (fm-parent <starting-point>)), but doesn't search <starting-point> and its children.

fm-kid-typed

```
(fm-kid-typed <self> <type>)
```

(Function) Finds the first <self>'s child whose type is <type>.

fm-descendant-typed

```
(fm-descendant-typed <self> <type>)
```

(Function) Finds the first descendant of <self> whose type is <type>

7.3 Misc

cells-reset

```
(cells-reset)
```

(Function) Resets the system.

8 Other resources

This tutorial just scratched the surface of cells. You can find more documentation about cells within the 'doc' directory in the source tarball or by looking at the source files within the directories 'cells-test', 'tutorial' and 'Use Cases'. A general overview of cells can be found in the file cells-manifesto.txt in the source tarball. You can also ask questions about cells on the project's mailing list: `http://common-lisp.net/cgi-bin/mailman/subscribe/cells-devel`