# Cells Overview

[document under construction]



by Kenny Tilton

# 1 Preface

## 1.1 The small picture

When a CLOS instance slot is defined to be a cell, that slot behaves much like its namesake, a spreadsheet cell. Consider a small but rich code excerpt:

```
(make-instance 'choice-radio
       :choice-manager <some radio group instance>
       :click-action #'(lambda (self click-event)
                         (setf (selection
                                  (choice-manager self))
                               self)))
       :selected (sm? (eql self (selection
                                  (choice-manager self))))
       :color-key (sm? (if (selected self) :green :red))
```

Some missing details, but enough to see how a cell-driven application works. Let's walk through the process by which a radio button gets selected.

When a radio-button style instance is created, it is associated with some other radio group instance which will manage all the choices available in that group.

The click-action is dispatched by code not shown, triggering a sequence of state changes, or *dataflow*: first the radio button sets the `selection` slot of the managing radio group to be itself, the clicked button. Unless this button was already the current `selection`, this constitutes a change in the value. When that happens...

...the dataflow engine looks to see if any other cells depend for their values on the slot `selection`. It discovers that each `choice-radio selected` slot depends on the group `selection` and triggers their re-evaluation.

The radio buttons' `selected` values may or may not change--some unselected choices stay that way. But two `selected` slots changing between true and false in turn trigger re-evaluation of their buttons' `color-key` slots, which change between red and green.

Something else happens when a cell changes value. *Before* other cells get re-evaluated, the engine invokes a generic callback function:

`(sm-echo-slot-value` *`slot-name instance new-value old-value`*`)`

...for any special handling the programmer chooses to specify. In this case a specialization on `(slot-name (eql 'color-key))` would tell the window manager to redraw the choice widget so the new color gets manifested. End of story.

## 1.2 The big picture

Cells support a declarative paradigm in which the motivating force behind an application's overall behavior is dataflow through a dependency graph maintained automatically by the system. The benefits are several.

- Eliminates the class of bug in which program state is internally inconsistent.
- A corollary benefit: eliminates the housekeeping burden of keeping consistent the state dependency graph implicit in any application.
- Makes manifest the state dependency graph implicit in any application. Actual runtime dependencies can be graphed.
- Program semantics are fully encapsulated by the declarative model; the rule for a cell defines it completely.
- Cells let us tailor instances with rules, not just literals, making objects more reusable. For example, some GUI schemes allow widgets to specify their dimensions with options such as :sticky-left or :fixed or :elastic-right. With cells you can write any algorithm you like to decide a widget's geometry. Furthermore, by arranging instances in a navigable namespace, those algorithms can draw on arbitrary application state.
- Divides and conquers application complexity. No cell in even the most complex spreadsheet touches more than a handful of cells, usually just one or two. The complex solution emerges from multiple small but high-quality solutions to small problems. Our experience has been the same with Cells in application development.

## 1.3 Original motivation

Cells were developed originally to solve tricky GUI layout requirements of an arithmetic tutorial: simple things such as vertically center-aligned fractions, complex things such as long division problems. The goal was to format arithmetic nicely while students typed, so characters would be deleted as well as inserted. cut-and-paste, too.

In the case of the fraction, the numerator and denominator knew their own widths, but until they knew the width of the containing fraction they could not center themselves. The fraction could not decide its width without knowing the width of each part. There was no circularity, but no one component could decide all of its geometry without some information from another component; a precise order of calculation of several slots from several instances was required. This order would be different for each type of arithmetic problem and expression. And, as an example, the numerator of a fraction can be another fraction or an expression under a radical or any of several other possible forms; a daunting combinatorial explosion was anticipated.

Cells solved that trivially because of its declarative, slot-oriented, just-in-time calculation approach. We still coded rules for slots of fractions one way and long addition another, but we did so in a declarative style and let the dataflow engine order the calculations implicitly by calculating cell slots on demand.

### 1.4 Scaling down

Out of curiosity as to just how good they were, we then applied cells to a deliberately trivial task to see if they were too big a hammer for simple problems. Sometimes productivity tools fail to scale *down*, being too much trouble for straightforward tasks.

The trivial task we settled on was maintaining a radio group. Cells not only solved that with little fuss, but when we forgot to initialize the radio group to some starting value, the cell-based approach happened to do the Right Thing. We took that as a Good Sign.

### 1.5 Scaling up

# 2 Prior Art

**2.1 Spreadsheet applications**

**2.2 Bottleneck functions**

**2.3 Notification Schemes**

2.3.1 TCL Provider Classes

2.3.2 GoF Observer pattern

2.3.3 SmallTalk

**2.4 Constraint programming**

2.4.1 Sketchpad, 1962

2.4.2 Steele, 1980

**2.5 Garnet/Amulet**

**2.6 COSI**

**2.7 Frames, slots, daemons**

2.7.1 Quintus Prolog

http://www.aft.pfc.forestry.ca/Seidam_Documentation/Experts_Guide/reshell/Frames35.html

2.7.2 Lab 3

http://www.ida.liu.se/~TDDA23/ailabs/frames-en.html

2.7.3 PORK

http://www.ri.cmu.edu/pubs/pub_371.html

# 3 Introduction

## 3.1 In Brief

Cells let you define selected slots of CLOS instances as if they were cells in a spreadsheet. As with spreadsheet cells, a cell slot (cell, for short) can have bound to it a literal value or a formula. The formula is any Lisp form which, when wrapped in a suitable macro, enjoys access to anaphoric variables 'self (the CLOS instance of the slot) and, via the macro `cached-value`, the most recent value calculated by the formula.

If a rule dynamically accesses another cell, a dependency is automatically recorded. Each time a rule runs the dependencies are re-evaluated, so a branching rule may depend on different cells from time to time.

When a cell changes, all cells dependent on that cell are recalculated. This eager evaluation cascades recursively from one used value to its multiple users, recursively through the dependency graph until one of:

- terminating at a computing cell which recomputes the same value, despite the changed inputs, such as the many radio buttons that recompute "off" after each radio change.
- terminating at a computed cell which does in fact change value, but which value no one uses. Such a cell would exist for its echo, if not by mistake.
- if it loops back to itself propagating recursively to smUsers. no error is signal if the cell is defined to be cyclic (details below).

though if any recalculated cell happens to come up with the same value as before, propagation does not continue to its dependents.

The user may define callback echo functions to be invoked when a cell changes. Echo functions are called before propagation to other cells. For example, a cell for the color slot of a widget has an echo method which triggers a screen update so the new color gets shown.

Echo functions are the output of the overall model. Input to the model is achieved by setting special variable cells to information culled from OS events.

## 3.2 Some Details

Literals or formulas can be offered via :initform, :default-initarg or make-instance :initarg[1]. Think of this as instance-oriented programming, since different instances of the same class can have different rules for the same slot.

1. An open work item is getting update-instance-for-changed/different-class to work with cells.

If a formula consults, directly or through a function, another cell of any instance, a dependency between the two cells is recorded[1]. Cells are then automatically recalculated when their dependencies change.

Every model-object has an `mdName` slot. Included in the distro is a Family model with a cell slot for `kids` and a host of routines that let you manage a namespace of model-object instances. `kids` is itself a cell, so the model population itself can change dynamically in response to other changes in the model.

It is possible to define what we call echo functions. These are dispatched when a cell changes value. The default `eql` test for deciding if a slot has changed can be overridden by client code. Thus if a cell `myColor` changes from red to blue, an echo function can trigger screen redraw. (Our experience has been that it is a Bad Thing to have side effects inside cell formulas, though we have gotten away with it occasionally. Can you say “unsupported”?[2]) The echo function is passed the instance and the old and new values of the slot.

### 3.3 Different Cells

There are several kinds of cells besides the standard kind. Ephemeral cells revert to nil after propagating news of any new state to dependents, good for events.

Another kind is what I call a Drifter...unlike other ruled cells, you can `setf` their value; the formula for a drifter is understood to return a delta to be applied to the current value. I might use this to define a clock whose hands would normally move along with a cell feed from the system clock, but which could be reset at will, resuming their ruled motion once reset.

Others are Delta (when a numeric Delta "changes" from 3 to 0, it is considered "unchanged", i.e., the value is understood to be a change, and we are saying "if the change is 0 I do not need to be recalculated") and Stream, currently under development in support of a report writer...set a Stream cell to a list and when any dependendent formula is triggered to recalculate, the formula will see not the list but each element in turn as if by a series of client settings of a conventional cell.

More varieties are likely to be invented all the time. Every time cells are applied to a new project, new kinds of cells and Synapses (see next) get invented.

---

1. Sadly this compromises GC, so when you want to lose something from the model you have quiesce it with a call to not-to-be. I looked at weak hash tables at one point, forget what happened. I always manage model instances in a Family-based namespace, and the 'echo function for the kids slot explicitly snuffs lost kids, so there was no pressing need for the weak-hash table trick. that could be revisited if necessary.
2. Mr. Roberts Neighborhood

### 3.4 Synapses

More control can be had over dataflow by placing filters called Synapses on any given dependency within a formula. Synapses are fully user-programmable, unlike cells.[1]

One good example is a "sensitivity" synapse, which says "don't recalculate me until you have changed by this amount". This can be used for efficiency to avoid excessive calculation, or it can be used for the semantic value of not having the model be over-responsive to its input.

Synapses also modulate the value returned to the formula. Synapses sit between the using and used cells, mediating both the decision to trigger and the result returned. An example of the latter is a "delta" synapse, subtly but crucially different from a delta cell. Imagine one cell indicates speed, and I am building an accelerometer widget. I want a value that tells me the *change* in speed. Or maybe I want to trigger an airbag if the speed drops five percent in a few microseconds. Cell formulas out of the box just see the current state of the universe; to make a determination like those just described would require a lexical closure in which the formula could preserve state across invocations. Synapses make that a little easier, and only a synapse can avoid triggering of the formula in the first place.

### 3.5 Self-optimizing

Ruled cells optimize themselves out of existence if no cells accessed by the rule are associated with a cell. That needs some explaining. In support of this optimization, the programmer must declare if a cell *not* being assigned a formula will be constant or not. If I place a button at `(make-position 25 25)` it stays there. But if I specify the same value as being a variable cell (**CellVariable**), by coding: `(cellv (make-position 25 25))`, then later it can be moved.

By omitting the `cellv` macro I promise (and internals enforce) no changes will be made to a given cell. The dataflow engine uses this information to optimize things by not establishing a dependency on that slot when a rule accesses it. This happens to improve performance tremendously.

### 3.6 The trouble with cells: a paradigm shift

It takes a while to learn to think in terms of dataflow. Cells work differently than conventional code. Normally procedural code runs around changing internal application state and manifesting those state changes to the world outside. With cells, one sets up declaratively a domino effect in which change to one variable cell triggers a cascade of other changes, dispatching echo methods to manifest each such change.

In our case, even after many months of intensive use we caught ourselves regularly slipping into a procedural style. This would happen most often when adding a signif-

1. Plans are to make both cells and Synapses fully programmable.

icant new chunk of functionality, because once a subcomponent is expressed with cells, one is compelled to use cells to extend it. And that is another problem with cells....

Cells are hard to use in half measures. If A varies according to B and C, both of which vary at run-time, then you cannot make just A and B into cells. C must be made a cell or, when it changes, A will just sit there un-refreshed. The cell interface does not even include a function to force re-evaluation of a slot. If one were included, that still leaves the problem of tracking down all the places C might change and ensuring each includes code to refresh A. It is easier to go with the paradigm and just make C a cell, and this is why cells once applied tend to spread throughout the application.

This is not to say that one has to use cells for everything. A chess application built with cells could include a rule like this:

```
:next-move (sm? (massive-chess-engine-make-move
                  (^board *game*)))
```

The chess engine can be anything you like. But the current board position should be a cell.

# 4 Design Principles

## 4.1 Ease-of-use

Cells are interesting mostly because they make programming easier. It is no good having a productivity tool that is hard to learn or use. So the first design principle is that cells should be easy to use:

### 4.1.1 Do not make programmers worry about internals

In the early days a spreadsheet author could not reference a cell above the cell being defined. In effect they were being forced to deal with the internal top-down recalculation sequence. A number of times while developing cells we steered clear of decisions that would have required users to think about how cells work internally.

For example, as things stand now, the order in which users of X get notified of changes to X is an accident of the order in which they consulted X. Any problem arising from that unpredictableness was solved within the dataflow engine rather than force programmers to worry about ordering their slot accesses just so.

### 4.1.2 Syntactic simplicity

### 4.1.3 Automatic dependency tracking

### 4.1.4 Model population growth/contraction

## 4.2 Seamless integration with host language

Make new datatypes, not new languages.[1] Build on existing compilers, documentation, and standardization if any. Put energy into the new, not reinventing the old around something new. An important design principle is then to blend in with the host language as cleanly as possible, such as by working within CLOS.

## 4.3 Purity (e.g., no setf of ruled cells)

We could do it, we could setf an CellRuled, but it would be wrong[2]. The rule for that cell would then no longer define the slots semantics. Arbitrary other code are then imposing their semantics on the slot.

Design purity also motivates the commitment to eager evaluation.[3] Without eager evaluation the dataflow is not automatic and the paradigm then is broken; any code perturbing program state must also determine which lazy evaluations to kick off.

1. The bumper sticker “Make love, not war”
2. Richard Nixon to John Dean in the Oval Office re paying hush money to Watergate burglars
3. The idea that state changes get propagated fully as they happen. Lazy evaluation models simply flag dependent state as obsolete, deferring reevaluation until the state is next accessed.

On the other hand, below we list efficiency as another guiding design principle. It may well be that at some point we will want to add an option for lazy evaluation of selected cells. Used judiciously, manually keeping such cells up to date would be manageable.

### 4.4 Power

Purity aside, we do our best to support the unsupported, such as state propagation from within rules. In the rare cases where we have done that, if something goes wrong we first see if the system could have produced correct results anyway. If so, we change the system. (If not, we back out the unsupported code.) The goal is to let programmers code rules any way they like and still have it all work.

This principle has been important in bringing cells to where they are now. Early on when confronting ideas which on their face seemed unreasonable, such as managing the model population via cells, we always gave it a try. When it broke we investigated why. We then looked for a way the problem could be surmounted without compromising the overall design.

The alternative was to give the application developer something else to worry about. “Ah, well, unfortunately that won’t work. What you have to do is...” To date many seemingly unreasonable demands on the dataflow engine have been made to work, making cells more powerful and transparent than otherwise would be the case.

### 4.5 Efficiency

From the beginning cells have been groomed for prime time.[1] That explains the ease-of-use design imperative, and it explains the pervasive emphasis on efficiency. The engine dynamically prunes dependencies after each evaluation of a rule; if a branching rule ends up not accessing some state accessed during the preceding evaluation, that dependency is dropped. Rules that end up with no dependencies go away. Dataflow stops at any unchanged node. Synapses let expensive rules control how often they get run. Half a dozen implementations have been built in search of the fastest algorithm.

On top of that, cells begin with a head start over procedural code, because with cells the application has a dependency graph telling it exactly what work is required given any particular program input.

1. Saturday Night Live “Not Ready for Prime Time Players”

# 5 Conclusions

Cells have come to permeate our applications, including a cell-based GUI framework. Our subjective experience is that programming is much easier with cells. More functionality gets built in less time with fewer bugs. The improvement experienced in fact seemed to go well beyond what we would have expected from the obvious advantage of automatically keeping program state self-consistent. Rumination on this puzzle produced these tentative explanations:

5.1 Cells efficiently support a declarative, functional style of programming. The semantics of any ruled slot are completely encapsulated in its rule, making programs more comprehensible. The underlying dataflow engine automatically keeps ruled values consistent with their rules, making programs more reliable. The syntax is nearly transparent and the mechanism works automatically, so the benefits are had without the cost of extra programmer effort.

5.2 Cells allow greater reuse of objects because different instances of the same class can be parameterized with different rules, not just different literal values.

5.3 Cells reduce the complexity of applications by decomposing the overall application state complexity into manageable pieces. Again the spreadsheet is a good example: complex models arising from so many simple cells tapping one or two other cells. We suspect that, as applications increase in size and with them the number of interesting internal state variables, there is an exponential growth in the number of state interdependencies[1], explaining why small programs are easy to get right and very large programs often fail ever to be delivered.

5.4 Any application is a model With a dataflow engine, the model works by itself. Without an automatic state-propagation (dataflow) engine, the effort of making that model work falls on the programmer. "OK, that selects some text, now I have to enable the "Cut" menu item." Programmers are in effect hand-animating their models, as tedious and error-prone an activity as is hand-executing an algorithm.

5.5 A grander way to contrast dataflow with hand-animation: Cells add causal power to programming. Programming with cells is like setting up a pattern of dominos standing on end; the cascade produced requires just one tipped domino, the rest takes care of itself. The dominos *do* have to be set up properly, but then we are done. Cells feel the same.

1. One interesting thing about cells when used throughout an application is that they make apparent the state dependency graph implicit in any application. A simple GREP utility can tell you the number of rules, dependencies, echos and dataflow variables in an application. More accurately, perhaps, a runtime profiler can describe the graphs which actually arise during typical application use. We are curious if such metrics will correlate with cost of development.

# 6 Future directions

## 6.1 Echoing

One objective of a dataflow approach is to keep state self-consistent. Echoing exists in part to keep state outside the model consistent with model state; if the `hilited` slot of a widget is true, the widget should look highlighted on the screen. To achieve consistency echoing must be dispatched not just when a cell *changes* from one value to the next, but also when the slot takes on its initial value.

Part of what the macro def-sm-echo does for us is create an independent notation that an echo method has been specialized on the slot name in question. This is an area of active exploration right now, and that mechanism will likely be improved.

Why do we not just echo every slot at instance-initialization time? For efficiency. If a slot is not echoed it need not be evaluated until sampled. But how often does that come up? If a slot is not echoed, won't it be used by the rule of some slot which /is/ echoed?

## 6.2 GC impact

If weak hash tables were used to record dependencies, could we dispense with 'not-to-be? Do all CLs offer finalization of GCed instances?

## 6.3 New kinds of Cells

A first step is to export an interface allowing users to invent their own cells without modifying the package per se.

## 6.4 New kinds of Synapses

Same as with cells. Allow users to invent new Synapses freely.

## 6.5 Destructive Synapses

Synapses smart enough to set the value of their user directly, rather than just signal that the rule should be re-evaluated. For example, (fSetAddtive) would increment the user by the delta in the used amount.

## 6.6 Parallel processing

If a cell with multiple dependents changes, it can rapidly be determined if the dependents are independent of each other. i.e., If A <- (+ B C) and C <- (- D B), then C should not re-evaluated until B has been re-evaluated. But if no such interference (as I call it) exists, we could kick off the re-evaluation of both in parallel.

A useful exercise would be to hack the propagation code to determine what fraction of overall propagations could have been parallelized. Certainly the classic problem of huge numbers of widgets all watching the focus slot of the window to see if they are focused-p would benefit from this. That intolerable performance hit has been

worked around, but with parallel processing the natural solution would have performed satisfactorily (sparing me the workaround).