The article and presentation suggest that this is a new idea for financial contracts. If so, the insights should be beneficial to financial software. But as a general software design idea, they are not new. Transposed to the area of GUI design, the "bad approach" rejected at the beginning of the presentation (list all pudding types, for each of them compute sugar content, etc.) would mean devising every screen of an interactive application in its own specific way and writing the corresponding operations—display, move, resize, hide—separately in each case. No one ever does this. Any GUI design environment provides atomic elements, such as buttons and menu entries, and operations to combine them recursively into windows, menus, and other containers to make up a complete interface. Just as the pudding combinators define the sugar content and calorie count of a pudding from those of its ingredients, and contract combinators define the horizon and value sequence of a complex contract from those of its constituents, the display, move, resize, and hide operations on a composite figure apply these operations recursively on the components. The EiffelVision library (see the EiffelVision documentation at *http://eiffel.com*) is an example application of this compositional method, systematic but hardly unique. The article's contribution here is to apply the approach to a new application area, financial contracts. The approach itself, however, does not assume functional programming; any framework with a routine mechanism and recursion will do.

Interesting modularity issues arise not when existing combinators are applied to components of existing types, but when the combinators and component types change. The presentation indeed states: "Only if we add new combinators or new ingredients would we need to enhance `S`" (the sugar combinator). The interesting question is how disruptive such changes will be to the architecture.

The set of relevant changes is actually larger than suggested:

- Along with *atomic types* and *combinators*, we should consider changes in *operations*: adding a calorie count function for puddings, a delay operation for contracts, and a rotate operation for graphical objects.
- Besides such additions, we should include *changes* and *removal*, although for simplicity this discussion will continue to consider additions only.

Assessing the Functional Approach

The structure of the programs as given is simple—a set of definitions of the form:

```
0 (a)             = b_a,0                    [1]
0 (c (x, y, ...)  = f_c,0 (x, y, ...)        [2]
```

for every operation `0`, atomic type `a`, and basic combinator `c`. The righthand sides involve appropriate constants `b` and functions `f`. Again for simplicity, we may view the atomic types such as `a` as 0-ary combinators, so that we only need to consider form [2]. With `t` basic combinators (`on_top_of`, `hipped`…) and `f` operations (sugar content, calories), we need `t` × `f` definitions.