properties and operations in common, in the same way that puddings can be grouped into categories. Such groupings would help model and modularize the software, with the added benefit—if enough commonalities emerge—of reducing the number of required definitions. This requires, however, taking a new look at the problem domain: we must discover, beyond functions, the essential *types*.

Such a view will be at a higher level of abstraction. One can argue in particular with the fixation on functions and their signatures. According to the article (italics retained), "An *American option* offers more flexibility than a European option. Typically, an American option confers the right to acquire an underlying contract *at any time between two dates*, or not to do so at all." This suggests a definition by variation: either American options are a special case of European option, or they are both variants of a more general notion of option. Defining them as combinators immediately sets them apart from each other because of the extra Date in the signature. This is akin to defining a concept by its implementation—a mathematical rather than computer implementation, but still implying loss of abstraction and generality. Using types as the basic modularization mechanism, as in object-oriented design, will elevate the level of abstraction.

Levels of Modularity

Assessing functional programming against criteria of modularity is legitimate since better modularization is one of the main arguments for the approach. We have seen the presentation's comments on this issue, but here is a more general statement from one of the foundational papers of functional programming, by Hughes (1989), stating that with this approach:

> [Programs] can be modularized in new ways, and thereby greatly simplified. This is the key to functional programming's power—it allows greatly improved modularization. It is also the goal for which functional programmers must strive—smaller and simpler and more general modules, glued together with the new glues we shall describe.

The "new glues" described in Hughes's paper are the ones we have seen at work for the two examples covered—systematic use of stateless functions, including high-level functions (combinators) that act on other functions—plus the extensive use of lists and other recursively defined types, and the concept of lazy evaluation.

These are attractive techniques, but they address fine-grain modularization. Hughes develops a functional version of the Newton-Raphson computation of the square root of a number `N` with tolerance `eps` and initial approximation `a0`:

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

with appropriate combinators `within`, `repeat`, and `next`, and compares this version with a FORTRAN program involving `goto` instructions. Even ignoring the cheap shot (at the time of the paper's original publication, FORTRAN was already old hat and `goto`s despised), it is understandable why some people prefer such a solution, based on small functions glued