through combinators, to the loop version. Then again, others prefer loops, and because we are talking about the fine-grain structure of programs rather than large-scale modularization, the issue hardly matters for software engineering; it is a question of style and taste. The more fundamental question of demonstrating correctness has essentially the same difficulty in both approaches; note, for example, that the definition of `within` in Hughes's paper, yielding the first element of a sequence that differs from the previous one by less than `eps`:

```
within eps ([a:b:rest])  =  if abs (a - b) <= eps then b
else within eps [b:rest]
```

seems to assume that the distances between adjacent elements are decreasing, and definitely assumes that one of these differences is no greater than `eps`.‡ Stating this property would imply some Design by Contract-like mechanism to associate preconditions with functions (there is no such mechanism in common functional approaches); the proof that it guarantees termination of `eps` would be essentially the same as a proof of termination for the corresponding loop in the imperative style.

There seems to be no contribution to large-grain modularity or software architecture in this and earlier examples. In particular, the stateless nature of functional programming does not seem (positively or negatively) to affect the issue.

The Functional Advantage

There remains four significant advantages for the functional approach as illustrated in examples so far.

The first is notational. No doubt some of the attraction of functional programming languages comes from the terseness of definitions such as the above. This needs less syntactical baggage than routine declarations in common imperative languages. Several qualifications limit this advantage:

- In considering design issues, as in the present discussion, the notational issue is less critical. One could, for example, use a functional approach for design and then target an imperative language.
- Many modern functional languages such as Haskell and OCaml are strongly typed, implying the notation will be a little more verbose; for example, unless the designer wants to rely on type inference (not a good idea at the design stage), `within` needs the type declaration `Double → [Double] → Double`.
- Not everyone may be comfortable with the common practice of replacing multiargument functions by functions returning functions (known in the medical literature as RCS, for "Rabid Currying Syndrome," and illustrated by such signatures as `(a → b → c) → Obs a → Obs b → Obs c` in the financial article). This is a matter of style rather than a fundamental

‡ In citing examples from Hughes's paper we have, with his agreement, used modern (Haskell) notation for lists, as in `[a:b:rest]`, more readable than the original's `cons` notation, as in `cons a (cons b rest)`.