whether this is worth the trouble. Making the state available to functional programmers through monads is akin to telling your followers, after you convinced them to embrace chastity, that having children is actually good, if with you.

Is it really necessary to exclude the state in the first place? Two observations are enough to raise doubts:

- *Elementary* state-changing operations, such as assignment of simple values, have a clear mathematical model (Hoare rules, based on substitution). This diminishes the main benefit expected of stateless programming: to facilitate mathematical reasoning about programs.
- For the more *difficult* aspects of establishing the correctness of a design or implementation, the advantage of the functional approach is not so clear. For example, proving that a recursive definition has specific properties and terminates requires the equivalent of a loop invariant and variant. It is also unlikely that efficient functional programs can afford to renounce programmer-visible linked data structures, with all the resulting problems such as aliasing, which are challenging regardless of the underlying programming model.

If functional programming fails to bring a significant simplification to the task of establishing correctness, there remains a major practical argument: referential transparency. This is the notion of substitutivity of equals for equals: in mathematics, `f (a)` always means the same thing for given values of `f` and `a`. This is also true in a pure functional approach. In a programming language where functions can have side effects, `f (a)` can return different results in successive invocations. Renouncing such possibilities makes it much easier to understand program texts by retaining the usual modes of reasoning from mathematics; for example, we are all used to accepting that `g + g` and `2 × g` have the same meaning, but this ceases to be guaranteed if `g` is a side effect-producing function. The difficulty here is not so much for automatic verification tools (which can detect that a function produces side effects) as for human readers.

Maintaining referential transparency in expressions is a highly desirable goal. It does not, however, justify removing the notion of state from the computational model. It is important to recall here the rule defined in the Eiffel method: *command-query separation principle* (Meyer 1997). In this approach the features (operations) of a class are clearly divided into two groups: commands, which can change the target objects and hence the state; and queries, which provide information about an object. Commands do not return a result; queries may not change the state—in other words, they satisfy referential transparency. In the above list example, commands are `start`, `forth`, and (in the finite case) `finish`; queries are `item`, `index`, `count`, `before`, and (finite case) `after`. This rule excludes the all-too-common scheme of calling a function to obtain a result *and* modify the state, which we guess is the real source of dissatisfaction with imperative programming, far more disturbing than the case of explicitly requesting a change through a command and then requesting information through a (side effect-free) query. The principle can also be stated as, "*Asking a question should not change the answer.*" It implies, for example, that a typical input operation will read:

```
io.read_character
Result:= io.last_character
```