The following observations counterbalance some of this possible criticism:

- The functional examples come from industrial practice; specifically, a company whose business appears to rest on the application of functional programming techniques. The principal example—specifying sophisticated financial instruments—addresses complex problems faced by the financial industry, which current tools do not address well according to the presentation's author, an expert in that industry. This suggests that it is representative of the state of the art. (The first example—specifying puddings—is academic, intended only as a pedagogical stepping stone.)
- One of the authors of the article (S. Peyton Jones), also acknowledged in the presentation as coauthor of the underlying theoretical work, is the lead designer of the Haskell language and one of the most notable figures in functional programming, bringing considerable credibility. The paper used as a subsidiary example in the later section "Assessing the Modularity of Functional Solutions" has been extremely influential and was written by another leading member of the functional programming community (J. Hughes).
- In spite of the reservations expressed below, the solutions described in these documents are elegant and clearly the result of considerable reflection.
- The examples do not exercise the notion of changeable *state*, which would favor an imperative object-oriented programming style.

We must also note that mechanisms such as agents, which provide essential ingredients of the full object-oriented solution, were openly inspired by functional programming ideas. So the conclusion will not be a dismissal of the functional school's contribution, simply the observation that the object-oriented (OO) style is more suited for defining the overall architecture of reliable, extendible, and reusable software, while the building blocks may involve a combination of OO and functional techniques.

Further observations about the following discussion:

- Object technology as used here takes the form of Eiffel. We have not attempted to analyze what remains if one *removes* mechanisms such as multiple inheritance (absent in Java and C#), genericity (absent in earlier versions of these languages), contracts (absent outside of Eiffel except in JML and Spec#), or agent-style facilities (absent in Java), or if one *adds* mechanisms such as overloading and static functions, which threaten the solidity of the OO edifice.
- The discussion is about architecture and design. In spite of its name, functional programming is (like object technology) relevant to these tasks and not just to "programming" in the restricted sense of implementation. The Eiffel approach explicitly introduces a continuum from specification to design and implementation through the concept of seamless development. Implementation-oriented properties of either approach, while important in practice, will not be considered in any detail.