`my_contract.value`—without knowing what version of the routine is used, in what class, and whether it is specific to that class or inherited.

Thanks to commonalities captured by inheritance, the number of feature definitions may be significantly smaller than the maximum `t` × `f`. Any reduction here is valuable: it is a general rule of software design that repetition is always potentially harmful, as it implies future trouble in configuration management, maintenance, and debugging (if a fault found its way into the original, it must also be corrected in the copies). Copy-paste, as David Parnas has noted, is the software engineer's enemy.

The actual reduction clearly depends on the quality of the inheritance structure. We note here that abstract data type principles are the appropriate guidance here: since the key to defining types for object-oriented design is to analyze the applicable operations, a properly designed inheritance hierarchy will ensure that classes that collect features applicable to many variants appear toward the top.

There seems to be no equivalent to these techniques in a functional model. With combinators, it is necessary to define the variant of every operation for every combinator, repeating any common ones.

Extendibility: Adding Types

How well does the object-oriented form of architecture support extendibility? One of the most frequent forms of extension to a system will be the addition of new types: a new kind of pudding, pudding part, or financial contract. This is where object technology shines in its full glory. Just find the place in the inheritance structure where the new variant best fits—in the sense of having the most operations in common—and write a new class that inherits some features, redefines or effects those for which it provides its own variants, and add any new features and invariant clauses applicable to the new notion.

Dynamic binding is again essential here; the benefit of the OO approach is to remove the need for client classes to perform multibranch discriminations to perform operations, as in: "if this is a fruit salad, then compute in this way, else if it is a flan, then compute in that way, else ...," which must be repeated for every operation and, worse, must be updated, for every single client and every single operation, any time a type is added or changed. Such structures, requiring client classes to maintain intricate knowledge of the variant structure of the supplier concepts on which they rely, are a prime source of architecture degradation and obsolescence in pre-OO techniques. Dynamic binding removes the issue; a client application can ask for `my_pudding.calories` or `my_contract.value` and let the built-in machinery select the appropriate version, not having to know what the variants are.

No other software architecture technique comes close to the beauty of this solution, combining the best of what the object-oriented approach has to offer.