

Master's Thesis Proposal

“Object-Oriented Immutable Data Structures”

Robert Will

July 15, 2003

It is proposed to implement a set of immutable data structures in an object-oriented language. Two different benefits are explained: the conceptual simplicity of referential transparency and the possibility for correctness proofs and consistency checks with varying degrees of formality.

1 Context

A *mutable* data structure is one, that can be changed with update operations. An *immutable* data structure is one that can't be changed: every modifying operation will return a new instance, both instance can then be used *independently* from each other. (The implementation may allow the two instances to share memory, but this fact is completely hidden from the programmer who can always reason as if all the instances are completely independent.) In fact, the theory of Abstract Data Types, which is based on algebraic specification, does *only* treat *immutable* data structures. Mutable DS' on the other hand use *destructive update*: changing an instance destroys the old instance. An instance of an immutable data types corresponds to one mathematical object (like an integer), while a mutable data type is rather like a variable (that as a value which is a mathematical object). Thus we could call the immutable data types “value types” or “mathematical data types”. They are also more fundamental than the “variable types” which build up on them.

Object-oriented programming languages have made the implementation of reusable data structures particularly easy: one can program a data structure

once as a class and then create as many instances as needed. The programmer need only think of one instance at a time and the run-time system (with garbage collection) will care for the rest. An important consequence of the OO way to data structures are the *controlled side effects* and *aliasing*: a data structure can be shared between several parts of the program; if one part updates the data structure, the other part will see the update, too. The side-effects are *controlled* because the shared data structure can ensure some form of *consistency* (expressed in its class invariant). Updates can only happen via the client interface of the data structure. Aliasing does make formal reasoning about programs very hard, but it is needed to keep programs reasonably short and efficient. Furthermore it enforces the view-point (advocated by Bertrand Meyer) that objects are a model of the real world¹.

2 The two Benefits

2.1 The best from two worlds

Experience shows that aliasing and mutable data structures are a necessary evil to built complex software. Immutable data structures, on the other hand, are much more easy to use: one never needs to think about state, one can use any kind of algebraic laws and program transformations, programming becomes rather declarative.

Hoare[2] first proposed an imperative programming language with immutable data structures, but he dis-

¹Siehe dazu auch das Buchladen-Beispiel in meiner SA.

couraged an implementation of that language for several reasons especially efficiency. Rather he proposed to use the language for program development only, transforming programs to an efficiently executable subset of the language. Nowadays there seem to be several reasons, why the complete language should be implemented:

- It can be used for prototyping (where efficiency doesn't matter as much).
- Experience shows that the real efficiency bottle necks of programs can't usually be estimated in advance, but must be found through profiling. Consequently a prototype can be made efficient by refining only critical parts of it and leaving the rest in a simpler, more abstract form.
- The immutable data structures can be used in the *assertion layer* of a language, that is, for development and debugging purposes (can be switched off for production runs).
- Immutable structures allow more typeful programming. Consider the higher-order function `map` that applies a function to every element of a sequence. Applied to a mutable list it will replace every element with the function's value, and applied to the immutable sequence it will - of course - return a new sequence with the function results. In the first case the return values must be of the same type as the original values (since they are stored in the same structure). The immutable structure doesn't have that restriction, since a new sequence is created anyway.

Hoare's argument, that inefficient abstract data structures should be mostly used for program development instead of final programs, is still valid. However, as Bertrand Meyer observes, it is the goal of tools and languages to help people do things right, not to prevent them from doing things wrong[6]. In this sense a good mixture of immutable and mutable data structures will certainly enhance the expressive capability of a programming language — especially since immutable data structures are conceptually much simpler than many other programming

language features and they don't interfere with any other feature!

2.2 Practical Proofs of Class Correctness

I had the idea for this proposal after thinking about an article in progress by Bertrand Meyer with the title of this section[7]. His proof principles are on the one hand proof techniques for programs with pointers, which I won't discuss here; and on the other hand model-based specification: a data structure is assigned a mathematical value and each update operation replaces that mathematical value. The informal idea for this is already found in [4]. Meyer does his class proofs by introducing *model variables* to his classes. The specification is then expressed using those variables as we know it from methods such as VDM, Z, or B.

A critical point for the formalisation of such proofs is that one needs formalisations of those "model data types". Experience with functional programming and algebraic specification shows, that such formalisations are often already executable programs. A very good example are HASKELL's algebraic data types and the accompanying functions, which serve at the same time as specifications, implementations and lemmas for proofs. It is the lemmas that are also needed in proofs of class correctness. Even if those lemmas don't coincide with an imperative/object-oriented implementation of the mathematical data type, an implementation can be very useful. First, it provides a frame in the OO language to which algebraic axioms and lemmas can be added, and secondly, it can also be used for debugging instead of proof: like loop invariants can be checked as runtime assertions, model-conformance can be checked, too.

With the point from the last section, that the more abstract, immutable implementation is like a prototype, the proposed kind of assertion checking means that the more efficient implementation is *tested against its prototype*. Thus the same assertions are used for formal proof and for automatic tests.

3 Goal

It is the goal of the proposed thesis work to *design and implemented an object-oriented library of immutable data structures*. It must include the simple base structures *Sequences*, *Tupels* and *Sets*. In a referential transparent context stacks and other types of queues are only special cases of sequences. *Relations* and *Finite Maps* should also be included. But they don't have a straight forward algebraic specification and thus require some creative work.

Tupels and Sequences should be strongly comparable to standard functional implementations like in, e.g., the HASKELL Prelude. EIFFEL's tupels are taken into account, perhaps they're already what's needed. For further inspiration the student should refer to:

- the earliest formalisation of data structures in history[2, part II], which is still very good
- data structures in model-based specification, e.g., the B notation[1]
- libraries of functional data structures, e.g., EDISON[8]

The fact that immutable objects can be seen as the *current values* of mutable objects should be exploited by implementing conversion functions from mutable to immutable data:

- *Initialize* mutable objects with immutable values
- *Set* the value of a mutable object
- *Retrieve* the value of a mutable object as an immutable object

Those routines can be used to reason formally about class correctness *and* to mix mutable and immutable data structures in one program, making it potentially much simpler to construct and understand on the one hand, and more efficient on the other hand. The mutable data structures of the EIFFELBASE library[5] would offer a good starting point for this.

The *retrieve* routine (which could also be called "value" or "model" (as Meyer does)) of mutable data structures will be used to express the "representation

relation" (as in [4]) formally in the module (or class) invariant, thus the post condition of mutating operations can be entirely expressed with executable functions, which makes them runtime-testable assertions. Furthermore the correctness (relating to the abstract implementation) can be proved without further overhead.

4 Mutable Objects in Immutable Structures

The documentation of the JAVA standard library's `HashMap` contains the caveat (corresponding) "If the value of the *equals*-function of contained objects changes, the behaviour of the data structure is undefined". This points to a rather profound conflict between the theory of data structures which supposes that the objects in a structure (e.g., a set) have value semantics, whereas objects do have their own "technical" identity and changing values. ("Technical identity" means that the object-identity of the programming language (reference-based) is usually not what we need as equality in the data structure².)

This is an open question per se and I already have some ideas how EIFFEL's "expanded types" and multiple inheritance can be used to solve that problem. (Note: expanded types have value-semantics as opposed to the usual reference semantics of object-oriented programming. Just using expanded types everywhere is, however, not a solution, since immutable data structure work perfectly well together with references and aliasing, because no side-effects can happen. HASKELL uses references, too...)

5 Further Research

I only propose simple implementations of standard data types. As a follow-up it would, however, be interesting to port more advanced algorithms[8] to the object-oriented world. Especially the fact, that several different and independent instances of one data structure can share memory makes those persistent immutable data-structures generally useful.

²Was heißt Fachschlüssel auf Englisch?

References

- [1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, London, 3 edition, 1972.
- [3] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, October 1969.
- [4] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
- [5] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, Englewood Cliffs, 1994.
- [6] Bertrand Meyer. Helping people to do things right, not preventing them from doing wrong. *Eiffel Liberty Journal*, 1(1), 1997.
- [7] Bertrand Meyer. Towards practical proofs of class correctness. <http://www.inf.ethz.ch/~meyer/-publications/#Inprogress>, February 2003.
- [8] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.