The Role of Memory in Object-based and Object-oriented Languages

Eric G. Wagner
Wagner Mathematics, R1 Box 445
Garrison, NY 10524 USA

Abstract
This paper introduces a mathematical memory model appropriate for programming languages with both ground types and objects, and uses the model to explore a number of programming constructs related to an elementary inheritance, overloading, and class specification.

1 Introduction

This paper reports on some of the recent theoretical and practical results on programming constructs that came about as part of the continuing project to design, implement, and extend the programming language LD³ (=Language for Data Directed Design) introduced at the first AMAST conference [7].

The main idea promoted in this paper is that the proper context for talking about many aspects of object-oriented and object-based programming is imperative rather than functional.

In the discussion of objects in Smalltalk-80 at the beginning of [1] they write,

An object consists of some private memory and a set of operations... .
A message is a request for an object to carry out one of its operations... .
The set of messages to which an object can respond is called its interface with the rest of the system. The only way to interact with an object is through its interface... .

This brief quote suggests several of the properties generally associated with objects:

1. Objects have memory.
2. Objects have an identity – sending a message to an object may change the contents of its memory but it does not change the object itself.
3. Objects are encapsulated, that is, you can only interact with them through a fixed interface and so the details of how the memory is structured and how the operations are implemented can not be exploited.

However, while the quote is suggestive neither it, nor the description of the things it suggests, are very precise. A closer look at the languages Smalltalk [1], Eiffel [3] and C++ [6] indicates that what is stored in the private memory of an object are references to other objects and that while the references stored in the memory can only be altered through the interface to the memory, it may still be possible to send messages to the referenced objects that will alter their memories. In these languages the “private memory” of an object may
consist of a set of instance variables or attributes whose values are references to objects. It is easy to define objects with the property that the memory of the objects referenced in their instance variables can be altered from outside the object. For example, (forgetting the "untypedness" of Smalltalk) we can define a class Link in these languages where an object of type Link has two instance variables next, and value of respective types Link, and Integer. Consider the situation where we have two objects A and B of class Link and \( A.next = B \) (i.e., in the content of the instance variable next in the object \( A \) is a reference to the object \( B \)). Sending \( B \) a message to change the contents of its value instance variable will have the effect of changing the memory of the contents of \( A \)'s next instance variable.

The above example also illustrates the idea that the form of objects can be recursively defined in the sense that a Link "has a Link in its memory".

Viewing the memory of an object as a tuple of instance variables is essentially equivalent to viewing it as an element of a product. It is not difficult to propose possible objects whose form does not fit this paradigm. An important class of such examples are those where the value of an object is naturally viewed as coming from a sum of products rather than from a single product. A simple example would be a BinaryTree where a BinaryTree is either an empty tree or a non-empty tree and non-empty trees have instance variables label, leftTree and rightTree. While this kind of structure can be simulated in Smalltalk, see [2], the simulation is awkward. We employ a framework here in which such structure are easily and cleanly definable. While these structures are reminiscent of those in ML [5], the fact that we are employing them in an imperative framework rather than in the functional framework used in ML makes a significant difference – see Section 4.

This paper supplies a mathematical model for such an imperative view of objects – a view that supports objects having an identity, objects being encapsulated, and objects having values that are drawn from a sum of products of sets of objects or ground values. Section 2 provides some mathematical preliminaries. Section 3 defines the notion of a class-basis – essentially an incomplete specification of a set of classes with defined and ground (primitive) types that does not include a specification of the methods. The section presents two equivalent models of states for a class-bases: a categorical model and an equivalent graphical model which provides a precise version of the familiar "pointer diagrams". Section 4 uses the first model of states to provide a mathematical description of the kind of state transformations desired as the semantics of methods and then uses the second model to provide examples of methods with such semantics. Finally, in Section 5, we show how the imperative framework provides a straightforward approach to simple inheritance and illustrate it with a number of examples.

2 Strings and Things

Let \( \omega \) denote the set of natural numbers, \( \omega = \{0, 1, \cdots \} \). For \( n \in \omega \) let \( \left[n\right] \) denote the set \( \{1, \cdots, n\} \), so \( \left[0\right] = \emptyset \).

For any set \( K \), a string of length \( n \) over \( K \) is a mapping \( s : \left[n\right] \rightarrow K \). Let \( \text{Str}_K \) denote the category of strings over \( K \) with, as objects, all strings over \( K \), and, as morphisms, \( \alpha : u \rightarrow v \), all triples \( \left\{ u, \alpha, v \right\} \) such that \( \alpha \cdot v = u \). Let \( K^* \)