Dylan Memory Library Overview

P. T. Withington

1. Introduction

The Dylan Memory Library allows more concise control over the allocation of memory resources representing objects than is provided by the base Dylan language. In particular, the memory library allows the creation of multiple *heaps* or *pools* for grouping object allocations; it provides for the control of the allocation and reclamation strategy of those pools; and it extends the Dylan object representation model to allow explicit control over object allocation and deallocation.

2. Concepts

The Dylan language specifies automatic memory management to facilitate development of applications. Often when an application is to be delivered, it becomes necessary to more finely control memory usage, to achieve performance goals. The Dylan Memory Library is designed to provide levels of finer control, without requiring the implementation of a complete memory management system by the programmer, while still retaining most of the safety of an automatic memory management system. (It is the case that some of the more sophisticated features of the library, if used incorrectly, can lead to the type of errors associated with any manual management system.)

In the Dylan language, the abstract concept of an *object* as a value is represented by the memory resources of the system where the application is running. Here we use the term object as a shorthand for both the abstract concept and the physical resources that represent it.

Pools, also sometimes called *heaps*, provide a way to group the representations of objects that share conceptual or representational attributes and assign an optimal managment strategy to the resources that make up their representation. The memory library specifies a range of pool classes that can be used to create pools tuned to particular situations. Objects are then allocated or created in a particular pool, according to the applications needs. Pools can be restricted to contain objects of only a single type.

Pools can be further subdivided into *locales*, which group objects spatially or temporily. Each pool has a default locale, where objects are normally allocated. Additional locales can be created to allocate objects in. When a locale is used for temporal grouping, various management strategies are available to declare an expected *lifetime* of the objects in the locale, to cause the manager to reclaim unused objects in the locale under program control, or to explicitly reclaim all objects in the locale at once.

3.0 The Memory Library

Using the Memory Library has the effect of enabling the init keyword of **make**, **memory-locale:**, which takes as argument an instance of **<memory-locale>** from which to allocate the storage for the instance under construction. Ordinarily, this keyword does nothing.

There are a number of memory pool classes, all subclass of **<memory-pool>**:

- **<managed-memory-pool>**, which has subclasses that handle automatically managed and manually managed memory pools.
- **<typed-memory-pool>**, which has subclass that handle memory pools all of whose objects have a single type.

The **<memory-locale>** class is used to group a subset of objects in a pool.

There are also a number of operations on objects and pools:

- **object-memory-pool**, which returns what pool an object is in.
- **object-memory-locale**, which returns what locale an object is in.
- **memory-size**, which computes the amount of storage that would be required by allocating a new object of a given class.
- **destroy**, which deallocates the storage for an object in a manually managed pool.

4. Open Issues

It should be possible to provide a simple persistent object pool. The basic idea is that you would associate a file with a pool and that file would become the backing store for the pool. The pool will have an additional operation that lets you set/get a “root” object (typically a table) to be able to re-find objects in the persistent pool. This is not intended to provide a general object database facility, but rather to provide a simple object-based storage mechanism for a standalone program to store something like “preferences”, defaults, etc. without the programmer having to resort to file I/O and dump/restore of data.

We should probably provide a finalization interface. Perhaps it should be generalized to support actions on other GC operations than just reclaim, e.g., it might be useful to be able to “hook” an object being moved/copied to a new locale/generation.

We should probably provide weak pointers