Jay Almarode

Multi-user canonicalization

DURING APPLICATION DEVELOPMENT, it is sometimes useful to guarantee that your code does not create two objects that are logically equivalent. Instead, you would like the attempt to create a new object to actually return an existing, equivalent object, if it should exist.

Otherwise, the new object can be created and registered in such a way so that subsequent attempts to create an equivalent object will return this one. This technique, called canonicalization, makes your code more efficient by eliminating redundant objects and allowing you to take advantage of object identity. For example, if you know that there will only be a single object to represent some logical entity, you can use identity comparisons (== or ~~) when scanning for the presence of the object in some collection. Identity comparisons are usually more efficient to use because they are typically in-lined by the compiler, and do not require fetching the objects to return an answer.

Most Smalltalkers are already familiar with the concept of canonicalization with the use of symbols. By definition, symbols are guaranteed to be unique, so that any symbol with the same sequence of characters will have the same identity. This means that no matter where or how a symbol is created, an identity comparison of two equivalent symbols always returns true. In fact, the implementation of = for class Symbol is the same as ==.

The uniqueness of symbols allows them to be used in fast identity-based collections, such as a key in an identity dictionary, while preserving the semantics of equality look-up. This is one reason why method selectors are symbols rather than strings, since they are used as keys in a method dictionary.

A common usage for canonicalization is to implement a smart cache of objects whose state is derived from an external system. For example, if objects are being materialized from a relational database, then a cache typically maps a relational primary key value to its corresponding Smalltalk object. If some part of the application needs an object with a particular key value, the cache is consulted first. If there is already an entry in the cache for that particular key, then the application can avoid having to execute time-consuming code to communicate with the relational database and perform the relational-to-object mapping, since it has already been done before (of course, if the relational data has been modified since the initial caching occurred, then the cache must somehow be updated or invalidated, but that is a different problem).

Building your own canonicalization mechanism is fairly straightforward in a single-user Smalltalk system. A typical implementation is to override the instance creation method to check for the presence of an existing, equivalent object before creating a new one. A common implementation is to maintain a dictionary in a class variable, where the keys of the dictionary are the logical values upon which equivalence is determined, and the values of the dictionary are instances of the class that have already been created. I suggest using a class variable, rather than a class instance variable, so that creating instances of subclasses consults the same dictionary. Another advantage of this implementation is that it is very easy to get all instances of a class and its subclasses.

To illustrate this technique, here is the implementation of an instance creation method for class Employee. In addition to having instance variables for name and social security number, Employee has a class variable called "CanonDictionary" that is initialized to a dictionary. In this model, social security number is the primary key upon which equivalence is based, i.e. we never want object memory to contain more than one instance of Employee with the same social security number. Since we always want an Employee to have a social security number, we override the "new" method to raise an error, and require instance creation to occur with the "name:ssn:" method listed here:

```
classmethod: Employee
name: aName ssn: aSSN
"Return an instance with the given name and ssn. If one
```

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

```
    does not exist in the canonicalization dictionary, create
    a new one; otherwise, return the existing one."
  ^ CanonDictionary at: aSSN ifAbsent: [ | emp |
        emp := self basicNew name: aName; ssn: aSSN.
        CanonDictionary at: aSSN put: emp.
    ]
```

This technique works fine in single-user Smalltalk systems, since only one user is creating objects in this image. But in multi-user Smalltalk, there may be concurrent users who are creating objects in a shared image.

This opens the door to the possibility that users will experience concurrency conflicts on the canonicalization dictionary. In addition, since each user operates with their own transactionally consistent view of objects, there may be more than one user who thinks he or she is creating the first instance of an Employee with a particular social security number. This is because neither user will see the other's modifications until his or her transaction is committed. At the very least, one of the users could experience a concurrency conflict, but it could be worse if both users were allowed to create logically equivalent instances of Employee and the application code depended upon their uniqueness.

Fortunately, by subclassing an existing specialized multi-user class, this situation can be handled correctly. In GemStone Smalltalk, the class RcHashDictionary provides concurrency semantics that are close to what is needed (see my column in the March–April 1995 issue of the *Smalltalk Report* for a description of reduced conflict classes). This multi-user dictionary allows concurrent updaters and removers from the dictionary to perform their operations without conflict, as long as they are using different keys. For example, two concurrent users who are performing at:put: operations with non-equivalent keys will not experience concurrency conflicts. But in our example, concurrent users might try to create instances with the same social security number, so they would experience conflict. What is needed is the ability to recognize these conflicts, choose one of the instances to be the canonical Employee with that social security number, and to replace all references to the noncanonical Employee with references to the canonical Employee (allowing the noncanonical Employee to be eventually garbage collected).

To solve this problem, I created a subclass of RcHashDictionary, called RcCanonicalDictionary. This class only needs to override one method to provide the desired behavior; however, to implement this method requires an understanding of how reduced conflict behavior is achieved. When a user attempts to commit a transaction, the underlying system detects if there are physical conflicts on objects, for example, checking if this transaction wrote an object that another concurrent transaction had already written and committed. For most objects, a physical conflict means the transaction cannot succeed. However, for special reduced conflict objects, they are given a second chance to determine if the physical conflict can logically be resolved.

This involves selectively updating the view of these objects so that the committed modifications of other users are visible, and then replaying the modifications of the current transaction on the reduced conflict objects. If the modifications can be replayed without failing, then the transaction is allowed to commit successfully.

For RcHashDictionaries, the method that replays updates to the dictionary is _replayAt:put:oldValue:. This method is similar to at:put:, except that the third argument is the original value at the given key before the update occurred (this argument is nil if the entry was added for the first time).

This allows the replay method to check if the value before the update is the same during replay as it was when the operation was originally invoked during the transaction. When the operation is replayed, if the current value is not the same as the old value, then we know some concurrent user has updated the dictionary at this key and we should fail the attempt to commit the transaction.

For our new RcCanonicalDictionary, rather than fail the transaction when another user commits a new entry at the same key, we would like to forget the value we were going to insert, and use the value that another user already inserted. This involves *swizzling* all references to the value we were about to insert to the new value inserted by a concurrent user. Fortunately, this is not very hard to do, since we can get a collection of all objects that were written during the transaction, and scan them to find references to our value. This avoids having to scan all of object memory to find references, which is prohibitive for a large scale number of objects.

One thing that must be accounted for when swizzling object references is to correctly update collections where the position of an object in the collection is dependent upon the identity of the object.

In GemStone Smalltalk, Bag and its subclasses use the identity of its elements to determine their positions in the internal implementation structures. Consequently, rather than overwriting the reference to the old value in these collections, the swizzling method first removes the old value and then adds the new value to the collection. Below are the methods to replay the insertion into an RcCanonicalDictionary when a physical conflict is detected, and the methods to swizzle references in general objects and for Bags.

```
method: RcCanonicalDictionary
_replayAt: aKey put: aValue oldValue: oldValue
"Stores the key/value pair in the dictionary. If there is
 already a value for the given key, then this method
 swizzles references to refer to the existing value."
| existingVal |
"see if there is now an existing entry (added by a
 concurrent user) "
existingVal := self at: aKey otherwise: nil.
"if there is no existing entry, update the dictionary;
 otherwise swizzle "
existingVal isNil
```

```
    ifTrue: [ self at: aKey put: aValue ]
    ifFalse: [
        " for each object written during this transaction,
         swizzle references "
        (System _hiddenSetAsArray: 9) do: [ :obj |
            obj _swizzleReferencesFrom: aValue to:
            existingVal
        ]
    ].
" return true to indicate that the transaction can proceed "
^ true
%

method: Object
_swizzleReferencesFrom: obj1 to: obj2
"Scan the named instance variables and indexable portion
 of the receiver, looking for references to obj1. For any
 that are found, replace the reference with obj2."
" first scan named inst vars "
1 to: self class instSize do: [ :j |
    obj1 == (self instVarAt: j)
ifTrue: [ self instVarAt: j put: obj2 ]
].
" scan indexable portion if necessary "
self class isIndexable
    ifTrue: [
        1 to: self _basicSize do: [ :j |
            obj1 == (self _at: j) ifTrue: [ self _at: j put: obj2 ]
        ]
    ]
%

method: Bag
_swizzleReferencesFrom: obj1 to: obj2
"If obj1 is contained in the receiver, remove all
 occurrences of it, and add the same number of
 occurrences for obj2."
" invoke superclass method for named instance variables "
 super _swizzleReferencesFrom: obj1 to: obj2.
(self includes: obj1)
    ifTrue: [
        (self occurrencesOf: obj1) timesRepeat: [
            self remove: obj1.
            self add: obj2.
        ]
    ]
%
```

Canonicalization of objects is a useful technique with many applications. In a multi-user environment, canonicalization mechanisms must take into account concurrent users creating equivalent objects.

This column has demonstrated one approach for solving this problem using the power and extensibility of multi-user Smalltalk.