PCLOS Reference Manual

Andreas Paepcke
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304

paepcke@hplabs.hp.com

November 19, 1991

(c) PCLOS Copyright 1988, 1991 Hewlett-Packard Company (by Andreas Paepcke)

Permission to use, copy, modify or distribute this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Hewlett-Packard not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Hewlett-Packard make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

HEWLETT-PACKARD DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL HEWLETT-PACKARD BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Preface

This document combines three formerly separate manuals of as many successive PCLOS versions: 2.0, 2.1 and 3.0. The three-part structure reflects this. Part one contains the bulk of information about PCLOS, and it is indexed for easy reference. Parts two and three describe modifications that were made to the system over time. These include bug fixes, upgrades and modifications in response to user feedback. The reader should therefore pay attention to those, since they occasionally provide information that supercedes material in part one. Note that parts two and three are not included in the index.

Contents

Part I

PCLOS Version 2.0

1 Introduction

The purpose of this document is to provide the technical information necessary to use PCLOS. Separate reports are available to cover more detailed questions on architecture, rationale and references to related work [1, 2, 3, 4].

PCLOS is a system that provides persistent storage and other amenities to CLOS programmers. CLOS stands for Common Lisp Object System and is being evolved into a standard for adding object-oriented programming to Common Lisp. PCLOS provides persistence at the object and the class level. Not all instances of a class need to be persistent. Transient objects may dynamically be made persistent and vice versa. Objects and individual slots may be cached explicitly by the programmer. While PCLOS currently does not cache on its own initiative, it maintains knowledge about "dirtyness" of objects which may be used to implement automatic cache maintenance.

Important features other than the preservation of state include transaction-based rollback of object state for all persistent objects and associative search over the state of the object base. PCLOS allows for the use of multiple, very different databases within the same session. As more databases are supported, this enables users to make choices between different databases based on their functionality and performance tradeoffs. Currently a simple, single-user in-core database with transaction rollback is included with the system. This database can be saved to and restored from an archive. Also included is the interface to Iris, an HP-internal research prototype of an object-oriented, multi-user, disk-based database[1] [5]. A product version related to the Iris prototype is called *OpenODB*. It has been announced by Hewlett-Packard, but no PCLOS interface to this product has been constructed. Its interface is very similar to that of Iris.

The system attempts to be as non-intrusive as possible. The guiding principle for the design is transparency. Ideally programs should run with minimal change under PCLOS. The implementation deviates from this ideal whenever one of two situations occurs: efficiency considerations suggest the introduction of special features, or some database concept is so useful that it should find entry into the programming world. Caching is an example for the first situation, transaction management is an example for the second.

The following sections introduce the bare minimum that makes understanding the later sections easier. Some of the information here might become more clear in those later sections when concrete operations are introduced. Any small database-specific pieces of information will be provided when the various standard operations are introduced. In addition, there is one section devoted to each kind of database where special capabilities and idiosyncracies will be described.

[1] See part II of this document for a third database to which an interface is included.

1.1 Architecture

Most of the control over object persistence is done through messages sent to PCLOS objects or classes. Some operations, however, do not affect particular objects, but control the underlying databases. This is where *protectors* come in.

Every database session is represented by an object called *protector*[2] which is obtained by the programmer through appropriate calls. This protector is the recipient of messages that explicitly control its associated database. Examples are transaction management and associative search. One object called the *master-protector* manages all protectors during a Lisp session.

Persistent objects are represented in memory by *husk* objects which do not contain any slot values unless the objects they represent are cached. Lisp *eq*ness is guaranteed for all objects. The programmer creates objects in memory in the usual way. When an object is to be made persistent, the programmer *protect*s the object by sending an appropriate message to it. At that point its in-memory representation is "gutted" – it turns into a husk object – and its state is transferred to the database. An appropriate database schema is automatically generated as part of this process. By providing methods with appropriate names, the programmer can operate on an object being protected shortly before and shortly after it is committed to the database. This can be used for cleanup or caching purposes. Transient and persistent objects are semantically and syntactically equivalent, except for the effects of concurrency control and, of course, life span. Slot accesses to persistent objects are transparently intercepted and appropriate commands are sent to the database for information update and retrieval. Transient objects are not known to the database and cannot be accessed by other users. Section 4 explains much of this in more detail.

Caching an object means to fetch all its slot values from the database and to put them into the corresponding husk object. The system will subsequently use this in-memory representation until instructed to change back to the default mode of operation. At that point the programmer determines whether the database is to be updated from the in-memory values or whether those values should be abandonned.

When the user runs a search over the database and an object satisfying the query predicate is already represented in memory as a husk, the system will notice that fact. If, on the other hand, a previously unknown object satisfies the query, a husk is made for it. We will say that the object is "internalized". Again, the programmer may operate on the object immediately after it is introduced into the address space by providing a method with an appropriate name.

If objects are protected or internalized, their classes – as defined in the current memory image – must conform to the classes that were defined in the memory image from which the objects were first committed to the database. This is enforced by the system. The programmer will be told in which way the current class differs from the class the database assumes.

[2] The concept of a protector was first introduced in the DOOM system by Brian Beach and Jim Kempf [6, 7].

1.2 Protocol Adapters

The ability to use multiple different databases for an object base without changing programs is useful but could come with a cost: it potentially limits the amount of functionality that can be made available for the program layer to the level of sophistication of the least advanced supported database. Some users might want to forego the data-store independence and want instead to make use of special capabilities of one particular database. *Protocol adapters* fill this need. While every database must support a minimum *core protocol*, protocol adapters may be added by PCLOS implementors to make available special, database specific features in some pleasing way. Each protocol adapter represents one database concept which might or might not be supported by any given database. Examples of protocol adapters are transaction management, complex queries, database archiving or support for the storage of particular datatypes. These protocol adapters are subdivided into *facets* which represent sub-aspects of the adapter's concept. Not all facets of a protocol adapter have to be supported by databases supporting that adapter. Examples of facets are rollback and concurrency control for the transaction adapter. The complex query adapter has facets that describe in which way the query is complex: are existential variables supported? Can results be multi-valued?, etc.

Each facet contains some machine-readable information that is used by the implementation. They also contain lists of *associated operations* which are methods that implement the facet. Both, protocol adapters and their individual facets also contain human-readable information that explains their capabilities. All of this information is accessible through the protectors.

It is possible to dynamically add and remove facets from adapters or adapters from protectors. Such a removal will cause the associated database interface to not support the associated concept. Removing the complex-query adapter from an Iris database will, for instance, limit the query capabilities of that Iris session to core-level queries.

Since not every database supports all facets of every protocol adapter, the user might attempt to execute operations that are not supported by the database being used. Database interfaces and adapters may therefore be associated with functions that are invoked when operations are attempted which require the presence of a particular unsupported adapter or facet. These functions are called *adapter-missing-actions* and *facet-missing-actions* respectively and may be installed and removed dynamically. An appropriate default action is taken if no explicit missing-action was specified.

2 Basics

2.1 Loading PCLOS

In order to load PCLOS, use `(require "pclos")`. No other action is necessary. It is assumed that "/lisp/modules/local/" is on the sys::*require-directories*. The load will look for the PCLOS software relative to the root of the PCLOS tree which is taken to be the value of the environment variable $pclos. This can be overridden, if global variable `pclos:*pclos*` is non-nil. It is then assumed to be a string pointing to the root of the PCLOS tree. Note that if you are using a CommonLisp dump which was made on another machine, that process will "know" about the environment variables that were defined on the other machine at the time the dump was created. If PCLOS seems to try the wrong places for loading software you should therefore try to see the Lisp image's 'idea' of the value of $pclos.

2.2 Preparing Classes

In order to make objects persistent, their class must have special capabilities. These are available to a class when its metaclass is "pclos-class". This is effected by including the option `(:metaclass pclos-class)` in the defclass statement. Slots in a persistent class may have one additional option: `:transient {T | NIL}` (default is NIL). If this option is T, the slot will not be given to the database and will not be restored by transaction rollbacks unless special measures are taken (see method `begin-transaction` on protectors).

Note that instances of persistent classes are not automatically persistent. Normal `make-instance` statements will produce transient objects that behave in standard ways. The object can be made persistent by sending the `protect` message to it.

Any module defining persistent classes should compile-load-eval-time require "pclos-load" and "pclos-meta". Any package should also use the package `:pclos`. All functions described below assume that this is true. PCLOS exports all symbols that are meant to be public.

Here is an example of the procedures explained in this section: we assume that the Lisp variable `prot` contains a protector object. Its creation will be explained later and is not important here.

```
(eval-when (compile load eval)
   ;; The compiler needs a small part of PCLOS for its work:
   (require ‘‘pclos-load’’)
   (require ‘‘pclos-meta’’))

(eval-when (load eval)
  (require ‘‘pclos’’))

;; Make your package get access to CLOS and PCLOS symbols:
(in-package :my-stuff :use '(:lisp :pcl :pclos))
```

```
(defclass CAR ()
  (
    (model       :initform NIL :type simple-string)
    (mileage     :initform 0)
    (current-rpm :initform 0 :transient T)
    )
  (:metaclass pclos-class)
  )
```

Note that some slots are typed while others are not. If a type is provided, PCLOS will do a more intelligent job of storing the slot values in databases. Some databases also allow more sophisticated searching for typed slots (see page 41).

The `current-rpm` slot is transient presumably because it will only be used for values without long-term significance.

```
(setf my-car (make-instance 'car
                            :model ``Mercedes 450 SEL''
                            :mileage 5))
```

The `my-car` object is transient, which means that no database contains it. But:

```
(protect my-car prot)
```

will make it persistent without causing any other changes. Messages to `my-car` will produce the same results as before the protection.

3 The Master Protector

Since work on databases is done through protectors, there must be ways to find protectors for given databases and to destroy them again. The object `*master-protector*` knows how to do this. In addition, individual protectors respond to some administrative messages which are documented in section 9.

Here is how you get a hold of a protector for one of the supported databases. This is the most common function performed by the master protector. Remember that you need to get a separate protector for each session:

```
(setf prot (find-protector *master-protector*
                           '(<db-type> <db-name>)
                           &optional (<db-specific> NIL)))
```

Before we explain the options to this method below, here are two examples of calls to `find-protector`:

```
(setf prot (find-protector *master-protector*
                           '(:iris ``smith@hplhoop:/databases/testdb'')
                           2000))

(setf prot (find-protector *master-protector*
                           '(:in-core `/users/jones/database/testdb'')))
```

Here are the details of the `find-protector` method:

```
<db-type>       == :iris | :in-core

<db-name>       == case <db-type>

                     :iris     A path to the Iris database. It uses the Arpa
                               notation.
                               Example: ``hoopla@hplhoop:/dbtests/zarmer''
                               This would either create or open an Iris db on
                               hplhoop, logging in as user hoopla. Rlogin criteria
                               hold for permissions.

                     :in-core  A string. If the in-core database will never
                               be saved to an archive, the string is used for
                               identification only. Otherwise it is also used
                               as the default file name for archives to which
                               the database will save and restore.

<db-specific>   == case <db-type>
```

:iris — This information determines the Iris log file size:
During transactions the database uses a file for the non-committed activities. Currently the length of this file is fixed, thereby limiting the size of transactions (see section on limitations). The number goes in chunks of 512-byte blocks and defaults to 1000 (i.e. 512000 bytes). A value of 2000 is generally good. This information is relevant only when finding a protector on a non-existing db, that is when an Iris db will be created.

:in-core — This information determines whether the system will attempt to restore the in-core database from an archive automatically as part of the find. The <db-name> parameter will be used as a path. If <db-specific> is T, a restore will be attempted and an error is signaled if the archive does not exist or is. inaccessible. If <db-specific> is NIL, no attempt is made to access any archive for restoring. If the parameter is omitted, the system will check whether there is a file <db-name> and whether it may be opened for reading and writing. If yes, the system will do a restore. Otherwise no error is signaled and the protector is returned.
Note that the restoration is only considered during the first find-protector call, that is when a protector has to be created. All subsequent calls simply return the existing protector.

Remember that you need to get a separate protector for each session. If you wish to use several databases simultaneously, you will want to get a protector for each one. The call above is not the only chance to fill an in-core database with information from an archive. Separate operations are available to do that. They will be introduced in section 13.

If an Iris database is created as a result of this call, there will be roughly one minute of wait time.

Here are some more obscure services provided by the master protector which may be skipped on a first reading but come in handy for real work. Sometimes it is desireable to cause the system to forget about all persistent objects managed by a protector. This situation arises most frequently during debugging, when it is expensive to leave a session and the programmer wishes to start over, possibly with some re-loaded application code. The following call will abandon all husk objects, all knowledge about in-memory classes which have been checked against classes known to the database and all other cached information. The database itself is not affected, that is no information is destroyed:

```
(destroy-protector *master-protector* <protector>)
```

Note that objects which were persistent through `<protector>` will be **unusable** after this operation. All slot values will have been destroyed.

It is sometimes desireable to avoid having the system recheck all classes but still to discard all husk objects. The method `abandon-known-objects` on protectors accomplishes this. It can be considered a less radical variant of `destroy-protector` and is documented on page 30.

Normally the code for supporting particular databases is loaded on demand, that is when the first protector for this type of database is found. The advantage of this is that no unnecessary code is loaded. The disadvantage is the delay incurred during the first `find-protector` operation. Preloading database code can be done through the master protector:

```
(load-database *master-protector* <db-type>)
```

The `<db-type>` parameter is the same as for the `find-protector` operation.

PCLOS redefines `sys:exit` to close all databases before leaving a Lisp session. In addition, each protector supports a `close-database` operation to close the database it represents. The master protector may be asked to close all known databases:

```
(close-all-databases *master-protector*)
```

When working with multiple databases it is sometimes desireable to find all protectors of a session, or to find all those that fill certain conditions. Example: find all protectors for Iris databases; or find all protectors on databases that are currently open. The following operations satisfy these needs:

```
(all-protectors *master-protector* &key (db-type T))
```

This returns a list of all protectors. If `db-type` is provided, it should be a database type as defined in the `find-protector` operation. In that case only the protectors representing the specified type of database will be returned.

```
(find-protector-if *master-protector*
                   test
                   &key (db-type T) (find-all NIL))
```

This call returns protectors based on the function `test` and the database type specification `db-type`. The master protector applies `test` to each protector. The `test` parameter must therefore be a function that takes a protector as its sole argument. It must return T or NIL, depending on whether the protector should be eligible for being returned. If `db-type` is provided, it should be a database type as explained in the `find-protector` operation. In that case only protectors representing a database of the specified type will be passed to `test` for consideration. Furthermore, if `find-all` is T, a list of all protectors qualified through this process will be returned. If it is NIL, only the first qualifying protector is returned.

Examples: Find a protector that represents a database with an open transaction:

```
(find-protector-if *master-protector*
                   #'in-transaction?)
```

Find a protector which represents an in-core database that has an open transaction and from which some objects are cached:

```
(find-protector-if *master-protector*
                   '(lambda (a-protector)
                      (and
                        (in-transaction? a-protector)
                        (get-cached-objects a-protector)))
                   :db-type :in-core)
```

Find all protectors whose associated database is currently open:

```
(find-protector-if *master-protector*
                   #'database-active?
                   :find-all T)
```

4 Controlling Persistence

Databases have knowledge about objects as well as their classes. The information about classes includes the names and types of slots and their allocations as well as the values of any class-allocated slots. Class persistence is therefore handled independently from object persistence, although the programmer does not need to be aware of this: whenever objects are protected, their classes are automatically protected also. Class-slot caching is also available through operations on objects. First-time users of PCLOS may therefore ignore explicit operations on class objects.

4.1 Object Persistence

To commit an object to the database, use the `protect` method on it. If methods `before-protect` and/or `after-protect` methods are provided which take an object as their sole argument, then these methods will be called immediately before and immediately after the protect. The `before-protect` may be used to fix up any slot values before they go into the database. The `after-protect` may, for instance, be used to cache objects as soon as they are protected. Note that these two methods will be called only once during the lifetime of an object.

When an object is internalized because it is in the result of a query, method `after-retrieve` is called with the object as its only parameter. This method may be used to cache objects automatically when they first come out of the database, or to set transient slots. The `after-retrieve` method may be thought of as analogous to an `initialize` method. It is called only once for each object that is brought in from the database during a Lisp session.

```
(protect <obj> <protector> &key
                           (protect-enclosed-objects T)
                           (run-after-protect-method T))
```

This will put the `<obj>` into the database represented by `<protector>`. If `:protect-enclosed-objects` is T, all objects recursively reachable from the slots of `<obj>` will also be protected. Otherwise, a non-recursive protect is performed. The operation returns a list of all objects that were newly protected.

Note that if a non-recursive protect is specified and unprotected objects are reachable from `<obj>`, an error is signalled because PCLOS does not store references to transient objects in the database. This option is intended for people who are intimidated by the machine doing something they have not planned themselves.

If `:run-after-protect-method` is specified as NIL, the `after-protect` method is not invoked.

If you protect an object whose class has never been "seen" by the database before, the protect may take much longer than if any object of this type has been protected before because an appropriate database schema is automatically generated in this case. Note also that if you protect an already protected object with :protect-enclosed-objects set to T, this call will not be a no-op because the system will have to check whether all other objects that are reachable from <obj> are also protected.

```
(make-protected-instance <protector> <class-name> &rest keylist)
```

This behaves like make-instance, but the resulting object is automatically protected as part of the call. The method is nothing more than a shorthand for the sequence `make-instance`, `protect`.

```
(unprotect <obj>)
```

Will remove the object from the database. Your memory-resident object will still be available. This is non-recursive. If you want to unprotect all objects reachable from `<obj>`, use:

```
(recursive-unprotect <obj>)
```

This will return the list of all objects that were newly unprotected.

NOTE:

> If other persistent objects reference unprotected objects, an error will be signalled when the unsatisfyable reference is accessed. The use of `recursive-unprotect` will properly unprotect all objects which are reachable from any slots of `<obj>`. But there might be other objects in the database which reference `<obj>`. PCLOS does not currently provide support for solving this problem.

Sometimes it is desireable to make a Lisp session forget that it ever internalized some particular object. This can be effected by:

```
(abandon <obj>)
```

This method affects only the in-memory Lisp session. The actual object in the database is not altered. Note, however, that the affected object is no longer usable after this method has been invoked on it. For a way to `abandon` all objects known to some protector, see method `abandon-known-objects` on page 30.

```
(recursive-abandon <obj>)
```

will recursively `abandon` `<obj>` and all objects reachable from `<obj>`. It will return a list of affected objects.

4.2 Class Persistence

As mentioned above, classes are automatically protected by the system when the need arises. This will be when the first object of an unprotected class is protected, or when the first object of an unprotected class is retrieved from the database through a search. In the latter case the class is obviously known to the database already. The class protect will in this case be limited to a consistency check. Explicit operations on classes are nevertheless provided to let programmers control when the overhead of class protection or consistency checking occurs. In addition, explicit operations on classes allow unprotecting of all instances of a class from one or more protectors.

Classes are themselves objects in a CLOS system and are obtained by using the CLOS method `(class-named <class-name-symbol>)` or `(class-of <object>)`.

```
(protect <class-obj> <protector>)
```

This call returns T if all went well.

```
(unprotect-instances <class-obj> <protector>)
```

This will unprotect all instances of the class on the given protector. The class itself will stay protected. Instances on other databases will also not be affected. The operation returns a list of all unprotected objects.

```
(unprotect <class-obj>
           &key
           (protector <home-protector>))
```

This will unprotect some subset or all instances of a class and will then remove all traces of the class from its "home protector", that is from the protector on which the class itself is protected. The subset of instances that are unprotected depends on the `protector` key argument. Remember that not all instances of a class need to reside in one database. If the `protector` is the home protector, all instances of the class on all protectors it knows about will be unprotected, which is generally what is wanted. If the specified protector is not the class's home protector, only instances on the specified protector will be unprotected before destroying class-specific information in the home protector. This operation returns T if all went well.

See also section 9 for unprotecting even more globally than at the class level.

5 Caching

You may load entire objects or individual slots into memory to speed up access to them. If such loading is done in a transaction, the item will be locked in the database and you have effectively *cached* it, else you should think of the loaded item as a snapshot from the database. We will use the word 'caching' for both of these cases. The mechanisms for caching slots and objects are independent, although there is an interaction: If an object is cached, all slots are treated as cached. See uncache and uncache-slot to understand how the mechanisms may be used independently.

5.1 Object-Level Caching

```
(cache <obj> &key
             (cache-class-slots T)
             (preserve-cached-values T))
```

If `:cache-class-slots` is T, the `<obj>`'s class-allocated slots will also be cached. If `:preserve-cached-values` is T, then any slots that were cached individually prior to the object cache will be preserved. Otherwise they will be overwritten. If you call cache on an already cached object, T is returned and nothing is done.

```
(recursive-cache <obj>
                 &key
                 (cache-class-slots T)
                 (preserve-cached-values T))
```

This is like cache, but it also caches all objects reachable from the object being cached. The operation returns a list of all objects that were newly cached.

```
(recache <obj>
         key
         (recache-class-slots T)
         (preserve-cached-values T))
```

If `<obj>` is not cached, this is equivalent to the `cache` operation. Otherwise this will update the already cached object to conform to the state of the object in the database, destroying all slot values of the previous in-memory copy.

```
(recursive-recache <obj>
                   &key
                   (recache-class-slots T)
                   (preserve-cached-values T))
```

Does recache to `<obj>` and to all objects reachable from it.

```
(uncache <obj>
         &key
         (uncache-class-slots   T)
         (uncache-cached-slots  NIL))
```

Write state of the cached object out to the database and destroy the in-memory copy of <obj>. All subsequent slot accesses will be to the database. If uncache-cached-slots is T, then slots that were cached individually will be uncached as part of the object uncaching.

Subtle point:

When writing the slot values back, PCLOS will protect any objects reachable from <obj> because transient objects may not be referenced by persistent objects in the database. So this operation can cause other objects to be protected as a side effect. The same is true for the write-back operation below.

```
(recursive-uncache <obj>
                   &key
                   (uncache-class-slots T)
                   (uncache-cached-slots NIL))
```

Uncaches <obj> and all objects reachable from it and returns the list of uncached objects.

```
(write-back <obj>
            &key
            (write-back-class-slots T))
```

Write state of the cached object out to the database but do not uncache the object. Retain the in-memory copy of the object state. Subsequent slot accesses will still use the copy in memory. The **subtle point** explained above in the context of uncache is relevant in the context of write-back also.

```
(recursive-write-back <obj>
                      &key
                      (write-back-class-slots T))
```

Execute a write-back operation on <obj> and on all objects reachable from it. A list of objects which were written back is returned.

```
(abort-cache <obj>
             &key
             (abort-cache-class-slots T)
             (abort-cache-cached-slots NIL))
```

Cause <obj> to be uncached without updating the database with the in-memory copy of the object state, that is, abandon the in-memory copy. If abort-cache-cached-slots is T, explicitly cached slots will also be affected.

```
(recursive-abort-cache <obj>
                       &key
                       (abort-cache-class-slots T)
                       (abort-cache-cached-slots NIL))
```

Perform an `abort-cache` on `<obj>` and on all objects reachable from it. Return a list of all the affected objects.

5.2 Slot-Level Caching

Slots are currently cached on a per-object basis. Although PCLOS does not currently support caching a slot for all of a class' instances, the programmer can produce that behavior by invoking the `cache-slot` method in a class' `after-protect` and `after-retrieve` methods. This will ensure that the desired slot is cached for all newly protected instances and for all instances that are retrieved through a search.

Caching of class-allocated slots **cannot** be done on a per-slot basis. See section 5.3 for how all class-allocated slots may be cached together in one operation.

```
(cache-slot <obj> <slot-name>)
```

Cache the specified slot. This mechanism is independent of the caching of the entire object.

```
(recache-slot <obj> <slot-name>)
```

If the slot is not cached, this is equivalent to `cache-slot`. Otherwise the operation updates the specified slot's value to conform to the current value of the slot in the database.

```
(uncache-slot <obj> <slot-name>)
```

Writes out the current value of the slot to the database and causes all subsequent accesses of slots to go to the database for the values.

5.3 Class-Level Caching

Programmers can live without any of the following class-level caching operations. This is because object-level caching will by default also cache class-allocated slots. The operations are provided to allow the caching of class-allocated slots without access to any instance:

```
(cache <class-obj> &key
                   (preserve-cached-values T))
```

This will cache all of the class-allocated slots of `<class-obj>`.

```
(recache <class-obj> &key
                     (preserve-cached-values T))
```

Will overwrite the class-allocated slots with the current values in the database. The slots will stay cached.

```
(uncache <class-obj> &key
                     (uncache-cached-slots NIL))
```

Write out in-memory values of all class-allocated slots and then cause them to be not cached.

```
(write-back <class-obj>)
```

Write the in-memory values of the cached class-allocated slots back to the database, but keep the slots cached.

```
(abort-cache <class-obj> &key
                         (abort-cache-cached-slots NIL))
```

Make all class-allocated slots considered not cached, but do not write out their values to the database.

6 Information about Objects and Classes

All PCLOS objects and classes can give useful persistence-related information. In addition, PCLOS classes and protectors can provide some information about persistent objects. This section introduces these operations. See section 11 for access to more low-level information.

6.1 Asking Objects

`(protected? <obj>)`

Returns NIL or non-NIL. A non-NIL result means that the object is protected.

`(cached? <obj>)`

Returns T or NIL.

`(protector <obj>)`

This returns the PCLOS protector on which the `<obj>` is protected.

`(describe <obj>)`

This offers information in addition to the standard `describe`: If the object is protected, information about the relevant database is shown, else the fact that the object is unprotected is announced. Information is provided about whether the object is currently cached. After each slot name a parenthesized expression of one or two characters is shown: "P" means that the slot is persistent. "T" means that it is transient. A "P" may be followed by a "C" which indicates that the slot is currently cached.

`(obj-dirty? <obj>)`

The PCLOS system keeps track of "object dirtyness". An object is considered dirty if it is not protected or if the object or one of its slots are cached and the cached information was modified. Several actions will cause an object to become clean. They are the obvious operations of protecting, uncaching, recaching, writing back or cache aborting. This "dirt-management" may be used in conjunction with the `get-cached-objects` operation to implement automatic caching.

6.2 Asking Classes

`(get-cached-objects <class-obj>)`

Returns a list of all instances of the class that are currently cached on any of the protectors on which the class keeps instances.

`(protected? <class-obj>)`

Returns NIL or non-NIL. A non-NIL result means that the class is protected.

`(cached? <class-obj>)`

Returns T or NIL.

6.3 Asking Protectors

In addition to the following operations, section 7 introduces an operation to find out whether a protector is in a transaction.

```
(get-cached-objects <protector>)
```

Returns a list of all objects that are currently cached on the specified `<protector>`.

```
(get-cached-classes <protector>)
```

Returns a list of all classes that are currently cached on the specified `<protector>`.

7 Transactions

Transactions serve two purposes: First, while in a transaction, locks are placed on objects that are accessed to control concurrency for databases that support multiple users. Second, a transaction may be aborted, thereby undoing whatever was done to persistent objects since the transaction was begun. These two aspects do not need to be supported by all databases. The Iris database supports both, the in-core database only supports rollback.

It is important to understand that a rollback also undoes "systemy" things: `Protect`, `cache`, `unprotect` or any of their relatives are undone like anything else.

Objects that are cached are protected by transactions in the sense that rollback is possible. This behavior may be overridden by the user (see function `begin-transaction`). This service is expensive in space, but no costs in time will incur outside of the `begin-transaction` and `abort-transaction` calls. The space costs amount to roughly the size of the affected objects.

It is also important to realize that symbols will be restored if they are referenced in any cached objects. The treatment of symbols in the context of rollback is different depending on whether the symbol is referenced from a cached or an uncached object: If the symbol is referenced from a cached object, a rollback will completely restore the symbol. This means that `defun` or binding of such symbols will also be undone. The rollback for cached objects is therefore somewhat more complete. The one aspect that is never undone for symbols is interning. If a symbol is interned during a transaction, it will not be uninterned.

As a general rule, transient slots are not restored by rollbacks, that is, their values are unchanged after a rollback. By setting a key-parameter in the `begin-transaction` call, transient slots may be protected for cached objects.

See the section on limitations to understand peculiarities regarding *eq*'ness.

```
(begin-transaction <protector> &key
                               (include-cached-objects T)
                               (include-cached-slots T)
                               (save-cached-transient-slots NIL))
```

This operation begins a transaction. It is an error to do this while already in an open transaction.

If `:include-cached-objects` is T, then rollback is supported for cached objects. This option is available because of the expense in storage of rollback support for cached objects.

If `:include-cached-slots` is T, then rollback is supported for cached slots.

If `:save-cached-transient-slots` is T, then rollback will be supported for transient slots of cached objects. Please note the implication of this: unless an object is cached, there is no way to roll back transient slots. Note also that by default transient slots even of cached objects are not rolled back.

```
(end-transaction <protector>)
```

Make all changes permanent in the database and release all database locks. If the protector is not currently in a transaction, this operation is a no-op.

`(abort-transaction <protector>)`

Cause rollback and release all database locks. If the protector is not currently in a transaction, this operation is a no-op. Note that Lisp *eq*'ness is maintained only for objects and symbols, not for any other items. Example: At the beginning of a transaction slot foo contains list `'(a b c)`. The slot is modified and then the transaction is aborted. Now slot foo will again contain a list `'(a b c)`, but the list before and after the transaction will not be *eq*.

`(in-transaction? <protector>)`

Returns T if the protector is currently in a transaction.

8 Finding and Retrieving

Associative search for objects in the database by properties of their state is done through the protector of a database. The core-level protocol supports a very simple, Lispy-looking query language which understands the concepts of PCLOS classes and slots. Interfaces to some databases that support more powerful query capabilities are equipped with a complex-query protocol adapter. Currently the Iris database is the only supported database with a complex-query adapter. Please refer to the special section on the Iris database for details. The description of how to find objects below are guaranteed to work on all supported databases. When qualifying objects are found in the database which have not yet been internalized, they will be internalized as part of the query solving process. Corresponding classes that need protecting will also be taken care of at that time.

Limitations to note:

Core-level queries can currently only involve one PCLOS class at a time. No queries are possible across classes. An example of this would be "Find all objects of any type whose slot 'foo has value 10". This will not work for core-level queries even if it is guaranteed that all classes known to the database actually have a slot named 'foo.

Another limitation is that the query process does not yet completely understand PCLOS inheritance.

Example:

Class *bar* inherits from class *foo*. Finding "all objects of type *foo* such that some criterion is true" will only search over objects of type *foo*, although one might expect the search to also extend over the objects of the inheriting class *bar*.

The negation operator is not currently supported by the Iris database.

The Core-Level Query Language:

Operators are '=', '/=', '>', '>=', '<', '<=', 'and', and 'or'. The functions 'and' and 'or' are n'ary. All others take two arguments. Equality is the same as "eq'ness" in the case of symbols and protected objects. For other lisp items one should think of query language equality as analogous to the Lisp "equal". Please see the examples at the end of this section for more details.

`(find-one <prot> <class-name> <pred>)`

This returns the first object found in the database that satisfies `<pred>`. If no objects satisfies the predicate, the function returns NIL. If `<pred>` is T, all objects of class `<class-name>` qualify.

`(find-all <prot> <class-name> <pred>)`

This returns a list of all objects satisfying `<pred>`. If `<pred>` is T, all objects of class `<class-name>` qualify.

Examples:

```
(find-all prot
          'my-type
          '(= name "fred"))

(find-all prot
          'my-type
          `(or
            (> name "fred")
            (= obj-ref ,interesting-obj)))
```

The latter will find objects of type `my-type` whose slot called `name` is lexically greater than `fred` or whose slot called `obj-ref` is *eq* to the object `interesting-obj`. Note the use of backquoting to get `interesting-obj` evaluated to an object.

`(finder <prot> <class-name> <pred>)`

This should be used when it is desirable to examine the result of the search one object at a time. In order to use this function a transaction MUST have been started. Failing to do this results in an error. The result of this function is a "finder" object which accepts various messages to provide successive elements of the query result set:

`(next <finder>)`

Returns the next object from the result set represented by `<finder>`, or NIL if no results are left.

`(all <finder>)`

Returns the entire remainder of the finder's contents in a list.

`(more? <finder>)`

Returns T, if more objects are left in the result scan, NIL otherwise.

`(close-finder <finder>)`

Will discard any information left in the finder and free all database resources held on account of the query the finder was the result of.

`(close-all-finders-by-class <prot> <class-name>)`

Closes all finders whose underlying queries involved `<class-name>`. If `<class-name>` is NIL, all finders are closed which is equivalent to

`(close-all-finders <prot>)`

This function unconditionally closes all finders on `<prot>`'s database.

9 The Protector Object

Each PCLOS protector represents one database session to the programming language object system. The protector object is mostly needed by PCLOS-internal top level service functions. Some of these services are, however, potentially interesting to the PCLOS user and are therefore made available by protectors. The sections below document some of them and may be skipped on a first reading. Other sections also contain operations for accessing protector services which are extensive enough to warrant their own section.

Note that unless otherwise noted, all operations can be undone through transaction roll-back. To be very clear about this: You can only undo anything if you have executed a `begin-transaction` on the protector beforehand! The very few operations that cannot be undone will refuse to execute if the protector is currently in a transaction. They are also clearly marked in **bold face**.

9.1 Controlling the Underlying Database

`(close-database <protector>)`

This will cleanly close the database associated with the protector. It depends on the the underlying database whether this operation is necessary before leaving a session. The PCLOS system will automatically close all databases when a `(sys:exit)` is executed, so the programmer should generally not have to use this operation explicitly. Transactions should, however, be closed before system exit. Otherwise a continuable error will occur. A database may be re-opened at any point using:

`(open-database <protector>)`

This allows continuation from the point where the database was closed. This function does not have to be called explicitly by the user unless an explicitly closed database is to be reopened.

`(database-active? <protector>)`

Returns T if the associated database is currently open, NIL otherwise.

`(database-name <protector>)`

This returns the name of the underlying database.

`(database-type <protector>)`

This returns type of underlying database (i.e. `:iris` or `:in-core`, etc.).

9.2 Destroying Data

`(abandon-known-objects <protector>)`

This will cause `protector` to forget about all objects it has internalized. It will still remember the classes and no changes are made to the database. This operation is used mostly during debugging sessions when new application code is reloaded and the programmer wants to start with a more or less clean slate. The advantage to having the classes remembered is that the time overhead for protecting the classes and performing consistency checks is avoided. For a completely clean slate use the `destroy-protector` method on the master-protector. Note that both of these operations will make all affected objects unusable, that is any pointers to them in the current Lisp session should be discarded.

```
(destroy-class <protector> <class-name>)
```

This operation brutally destroys the specified class and all its instances in memory and in the database. The class will still be usable in memory, but it will not be protected. Its class-allocated slots will be invalidated, and all instances will no longer be internalized or usable after the call.

```
(destroy-all-classes <protector>)
```

This operation brutally destroys all classes and all its instances in memory and in the database. The instances will no longer be internalized.

```
(unprotect-all-data <protector>)
```

This will `unprotect` all objects and classes the protector knows about. It will then **destroy all data in the database**. When all goes well, all objects and classes known to the current Lisp session before the operation will still be usable afterwards.

There is currently a problem with `unprotect-all-data`. If it runs into an object that references another object which it has already unprotected, the routine will break. The only "fix" to this is to manually destroy the class whose instance contains the bad reference. This is done using the `destroy-class` operation below. Then `unprotect-all-data` may be re-invoked. This obviously needs fixing.

```
(destroy-database <protector>)
```

Note: **This operation cannot be undone through transaction rollback**. This is the fastest way to destroy a protector's associated database and all persistent objects and classes known to the protector. In order to use the protector again after this operation, the programmer must execute `open-database` on the protector. That will create a new database and activate it for use. Note that all affected objects will be **unusable** after this operation and the database itself is completely destroyed.

9.3 Type Conversion Customization

This subsection describes how programmers may influence the way Lisp items are stored in the database. The most direct way to influence this is to type slots in PCLOS classes. PCLOS will attempt to map Lisp types to types that are native to any database used. Note that databases are sometimes stricter about type integrity enforcement than Clos.

When Lisp items are to be stored in a database that does not provide a corresponding type, some mapping must be found. PCLOS will take care of the following Lisp types:

- persistent objects.

- strings.
- lists (any cons).
- defstructs (I think, needs testing).
- integers.
- floats.
- symbols.
- simple-vectors.

Multi-dimensional arrays and array displacement are not currently supported. For programmers wanting to store items for which PCLOS provides no type mapping, there are two independent mechanisms available:

- The programmer my add or remove types that are assumed by PCLOS to be acceptable to the database directly, that is types that are considered database-nativedatabase-native.
- The programmer may provide conversion functions that take an unstorable item. Such functions return an equivalent item which is of a storable type and from which the original item can be reconstructed.

The operations concerned with these two mechanisms are introduced here. Note that all of these operations are on protectors which means that they are specific to each database session.

9.3.1 Managing Conversion Functions

```
(add-customer-prog-to-db-converter <protector>
                                   <key>
                                   <encode-func>)
```

This is used to extend PCLOS' ability to store complicated datatypes. A programmer may "teach" PCLOS about new types on a database by database basis. This is done by providing one or more functions which know how to encode a Lisp item into an item that can be stored directly on the database, that is into an item of a type which PCLOS already understands. Each such function should take one Lisp item and return two values: The second return value should be T if the function knew how to convert the Lisp item. Otherwise it should be NIL. If the function could do the conversion, the first return value should be the encoded value. A programmer may add as many converter functions as desired. The `<key>` parameter is used to find or remove functions after they were added.

Whenever PCLOS needs to convert a Lisp item to a database format, it will try the customer conversion functions in turn until one of them agrees to do the conversion or until no more converter functions are available. In the latter case PCLOS finds out whether it has built-in knowledge about the conversion. If not, it will signal an error.

The <encode-func> may be a Lisp function object (i.e. of the form '#foo) or it may be a function name. Note that if <encode-func> is a function object and the function is redefined, this redefinition will not be reflected in the function used for conversion. In that case it is necessary to remove the customer conversion function and add it back.

When the PCLOS system retrieves data from a database, it detects that an item has been "encoded" by a custom-function. It then passes the encoded item to a <decode-func> which must have been installed using:

```
(add-customer-db-to-prog-converter <protector>
                                   <key>
                                   <decode-func>)
```

This adds function <decode-func> to the chain of functions responsible for all conversions from database format to Lisp format of custom-encoded items. The <decode-func> should take an item encoded by one of the encode functions added through **add-customer-prog-to-db-converter**. It should return two values: The second result should be T or NIL depending on whether the function knew how to decode the information. If decoding was possible, then the first return value should contain the Lisp item which is the decoding of the encoded information. PCLOS will walk the decode function chain in the manner described for the encode function chain above.

All <encode-func>s should encode items in such a way that the <decode-func>s can recognize the type of the encoded information and take appropriate action.

To be clear: The <decode-func>s will only be called with items that have been encoded with an <encode-func>. The shape of the item produced by the <encode-func> is not used in any way to determine whether an item retrieved from the database is custom-encoded information. Separate tags are used for this purpose.

Example:

A Lisp compiled-function-item cannot be stored in a PCLOS database directly. An encoder function could encode this as a list:

```
(compiled-function <source-code>)
```

When the decode function is passed a two element list whose first element is **'compiled-function**, it could compile the second element and return the resulting compiled-function-item as a result.

Since multiple encode and decode functions may be added to a protector, one needs a way of finding the functions and for removing them:

```
(find-customer-prog-to-db-converter <protector> <key>)
```

This returns the customer encoding function that was installed under the specified <key> or NIL.

```
(find-customer-db-to-prog-converter <protector> <key>)
```

This returns the customer decoding function that was installed under the specified `<key>` or NIL.

```
(remove-customer-prog-to-db-converter <protector> <key>)
```

This removes the customer encoding function that was installed under the specified `<key>`.

```
(remove-customer-db-to-prog-converter <protector> <key>)
```

This removes the customer decoding function that was installed under the specified `<key>`.

9.3.2 Managing Native Types

Here are the operations which allow the programmer to declare that certain types should be considered native to the protector's database. Notice that if items of native types are embedded in non-native structures they will still be converted. Example: Lisp lists are considered non-native for the Iris database. If the slot of an object contains a list that includes an integer, that integer will be encoded as part of the list. If the integer is instead itself the value of a slot, it will be stored in an Iris database unchanged because integers are database-native for Iris databases.

```
(add-database-native-type <protector> <predicate>)
```

The `<predicate>` must be a function which takes any Lisp item and returns T if the item is of the relevant type or NIL otherwise.

```
(remove-database-native-type <protector>
                             <predicate>
                             &key
                             (test #'equal))
```

Removes the predicate that would allow some type to be considered native. Returns T if all right.

10 Protocol Management

Protocol adapters were introduced at the beginning of this manual. This section introduces questions the programmer can ask about the protocol adapters relevant for a protector's database and questions that may be asked of protocol adapter objects themselves. In addition, operations are introduced that allow the dynamic addition and removal of database interface features by manipulating adapters and facets. Parameter descriptions `<adapter>` and `<facet>` below imply adapter and facet objects respectively. Parameter descriptions `<adapter-id>` and `<facet-id>` mean Lisp symbols that are names for adapters and facets respectively.

10.1 Accessing Protocol Adapters Through Protectors

```
(add-protocol-adapter <protector> <adapter>)
```

This adds the specified protocol adapter object to the protector's database interface. The only way a programmer would get a hold of an adapter is by finding it or removing it from the protector first. The above operation can then used to add it back in.

```
(remove-protocol-adapter <protector> <adapter-id>)
```

Remove and return the specified protocol adapter, that is, cause the database interface to no longer support the specified protocol adapter. Note that removing a protocol adapter will cause the database interface involved to no longer support that adapter! This means that some operations on the database will fail.

```
(find-protocol-adapter <protector> <adapter-id>)
```

Return the protocol adapter object with the specified id or NIL.

```
(explain-protocol-adapter <protector> <adapter-id>)
```

Print the specified adapter's human-readable explanation to the output buffer.

```
(all-protocol-adapters <protector>)
```

Return a list of all protocol adapters supported by the associated database.

```
(explain-all-protocol-adapters <protector>)
```

Print the human-readable explanations of all the associated database interface's protocol adapters.

```
(explain-all <protector>)
```

Print the human-readable explanations of all facets of all the associated database interface's protocol adapters.

```
(add-adapter-missing-action <protector> <adapter-id> <action>)
```

Install `<action>` as the function to call when an operation is attempted that requires the presence of the specified adapter when that adapter is not supported by the database interface.

`(remove-adapter-missing-action <protector> <adapter-id>)`

Remove and return the function to call when an operation is attempted that requires the presence of the specified adapter when that adapter is not supported by the database interface. Return NIL if not found.

`(find-adapter-missing-action <protector> <adapter-id>)`

Find and return the function to call when an operation is attempted that requires the presence of the specified adapter when that adapter is not supported by the database interface. Return NIL if not found.

10.2 Controlling Protocol Adapters

We are getting to a pretty low level here. But a programmer can add, remove and find individual facets of protocol adapters and can have facets explain themselves.

`(add-facet <protocol-adapter> <facet>)`

Add the given facet object to the specified adapter. The only way a programmer can get a hold of a facet object is by finding it or removing it first from an adapter object.

`(remove-facet <protocol-adapter> <facet-id>)`

Remove and return the facet with the specified name from the protocol adapter. Note that removing a facet will cause the database interface whose adapter is involved to no longer support the facet of that adapter! This means that some operations on the database will fail.

`(find-facet <protocol-adapter> <facet-id>)`

Return the facet object with the specified name or NIL.

`(all-facets <protocol-adapter>)`

Return a list of all facet objects of the adapter.

`(explain-yourself <protocol-adapter>)`

Print human-readable information about the adapter to the output buffer.

`(explain-facet <protocol-adapter> <facet-id>)`

Print human-readable information about the facet to the output buffer.

`(explain-all-facets <protocol-adapter>)`

Print human-readable information about all of the adapter's facets to the output buffer.

`(add-facet-missing-action <protocol-adapter> <facet-id> <action>)`

Install `<action>` as the function to call when an operation is attempted that requires the presence of the specified facet when that facet is not part of the adapter.

`(remove-facet-missing-action <protocol-adapter> <facet-id>)`

Remove and return the function to call when an operation is attempted that requires the presence of the specified facet when that facet is not part of the adapter. Return NIL if not found.

`(find-facet-missing-action <protocol-adapter> <facet-id>)`

Find and return the function to call when an operation is attempted that requires the presence of the specified facet when that facet is not part of the adapter. Return NIL if not found.

`(explain-yourself <facet>)`

Print human-readable information about the facet to the output buffer.

`(associated-operations <facet>)`

Return a list of the names of methods that are involved in implementing this facet.

11 Debugging

If the global variable `pclos::*database-trace*` is set to T, all database operations will print debugging information to the output buffer.

`(database <prot>)`

Returns the database object (`<db-obj>`) of `<prot>`. This may be asked for its name through `(name <db-obj>)`. It is also used for the functions `display-xxx` described below.

`(pclos::internal-representation <obj>)`

This will print the contents of the object's internal representation. Similiar information is shown by `(describe <obj>)`, but while `describe` goes through normal slot access methods, `internal-representation` will show the contents of the actual storage used and will show some normally hidden administrative information.

`(pclos::display <db-obj>)`

Displays the contents of all db tables which have been accessed in this session. In particular, it shows the MASTER-CLASS-TABLE that contains information about PCLOS classes that are known to the database.

`(pclos::display-all-tables <db>) [Iris only]`

Goes into the database and finds and displays all tables (each corresponds to one PCLOS class).

`(pclos::display-one-table <db> <class-name>) [Iris only]`

Displays the table corresponding to the given class.

`(pclos::display-table-names <db>) [Iris only]`

Prints names of PCLOS classes represented in the database and known to the current Lisp session.

12 The Iris Database

This section presents information that is specific to the Iris database and did not fit well into other sections.

12.1 Complex Queries

The Iris database allows for much more elaborate query facilities than are available through the core-level protocol queries. These capabilities include multi-valued results, non-object results, searching over multiple classes, existential variables and the use of user-defined or database-intrinsic database functions. Each query consists of either two or three parts. Each part is introduced by a keyword:

- `:return` Introduces the list of return variables and their types.
- `:with` Introduces existential variables and their types.
- `:where` Introduces the search predicate.

The second of these query terms is optional. Return variables and existential variables are collectively called the *query variables*. Unless otherwise noted, all specifications are Lisp `symbols`.

12.1.1 The `:return` Term

The `:return` term controls how many values are returned for each query result. If, for instance, the `:return` clause contains 3 specifications, then every query result will be a three-element list. Each specification is a list of two symbols: A variable name which is used in the search predicate, and a Lisp datatype. If the search will bind the variable to an object slot that was explicitly typed, the datatype in the specification should match the slot's declared type. The value T should be used for untyped slots. All return variables introduced in the `:return` query term must be used in the search predicate.

12.1.2 The `:with` Term

The `:with` query term introduces existential variables. An existential variable in this context is an item which is used in the query predicate, but which is not returned in the query results. In this respect it behaves a little bit like a local variable. But there is an additional property of existential variables that deviates from this analogy: During query solution an existential variable is conceptually bound to every possible value of its associated type. All existential variables introduced in the `:with` query term must be used in the search predicate.

12.1.3 The `:where` Term

This term introduces the actual search specification. It contains conjunction and disjunction — introduced by `and` and `or` — and other operators. All operator names are assumed to refer to functions known to the respective database, except for the name `slot-value`, which has a special meaning for PCLOS query preprocessing and should be considered reserved. `Slot-value` is a pseudo-operator which takes two arguments: a query variable that refers to an object and a slot name. The `slot-value` expression refers to the specified slot of the specified object during the search.

If the search specification is T, all objects introduced in the `:return` term are returned.

Note that even with complex queries it is not possible to search over anything "deeper" than direct slot values in the sense that one cannot search for objects that are enclosed in lists or vectors.

12.1.4 Complex Query Examples

Assume the following scenario: Classes `animal`, `ranger` and `cares-for-link` are defined as follows:

```
(defclass ANIMAL ()
  (
   (species)
   (age :type integer)
   )
  (:metaclass pclos-class)
  )

(defclass RANGER ()
  (
   (name :type symbol)
   (age  :type integer)
   )
  (:metaclass pclos-class)
  )

(defclass CARES-FOR-LINK ()
  (
   (left)   ;; Always contains a ranger.
   (right)  ;; Always contains an animal.
   (hours-per-week :type integer)
   )
  (:metaclass pclos-class)
  )
```

Assume further that <*mary*> is a 23-year-old ranger whose name is Mary, that <*tiger*> is a 10-year-old animal and that <*camel*> is a 30-year-old animal. We postulate further that we have two `cares-for-links`: One link indicating that Mary cares for <*camel*> 30 hours per week and one indicating that Mary cares for <*tiger*> during 10 hours of each week. Figure 1 illustrates this scenario.

Here are some examples of possible queries:

Find all animals whose age is greater than 9:

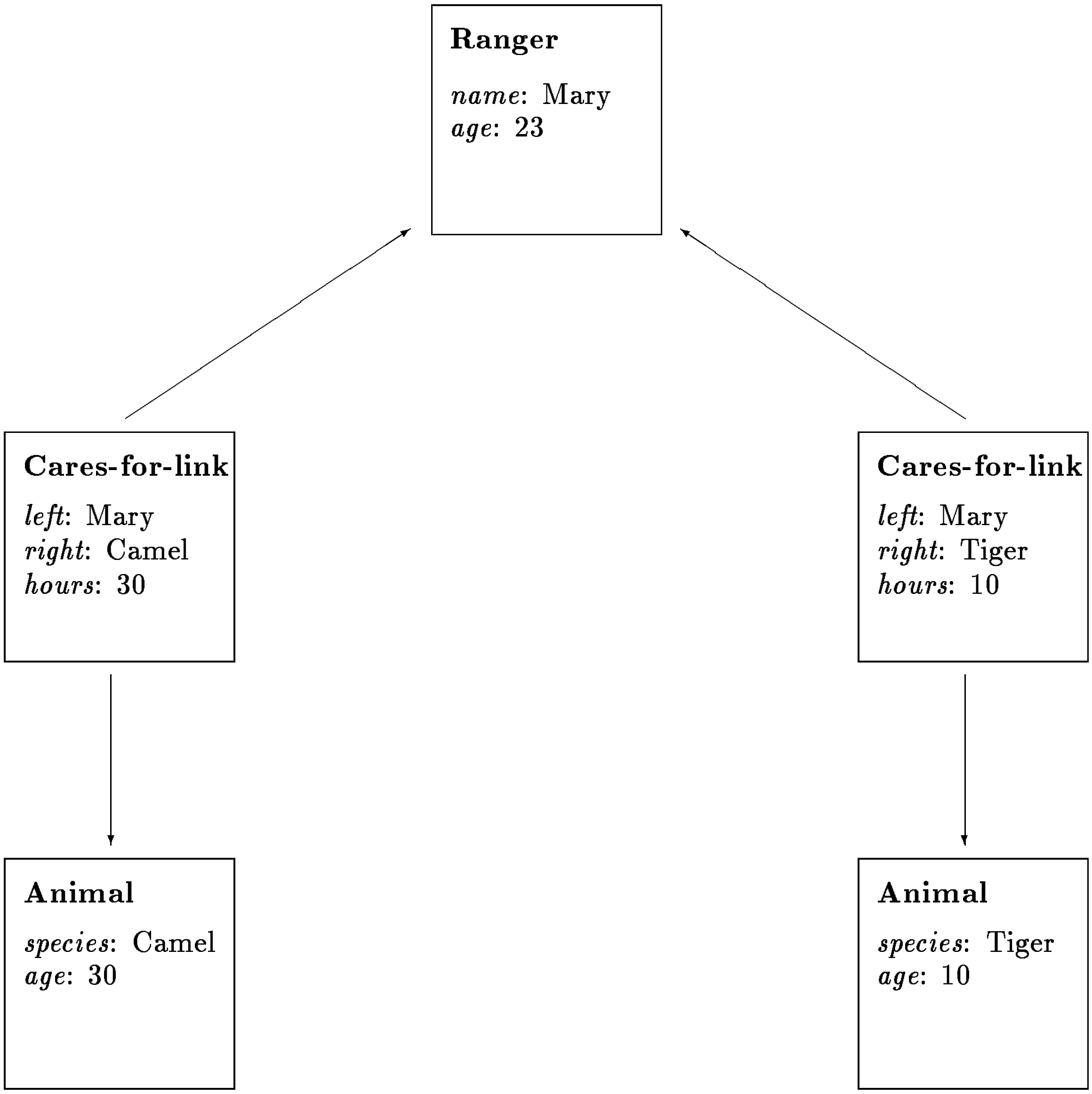

Figure 1: Scenario for Searching Examples

```
(find-all *prot*
  :return
  `(
    (the-animal animal)
    )
  :where
  `(> (slot-value the-animal age) 9)
  )
```

Returns: ((<*tiger*>) (<*camel*>))

Note in this context an important problem: If the `age` slot for `animals` had not been declared to be of type `integer`, the '>' operator would have caused an error because Iris has static type checking. It would have noticed that *anything* could come out of slot `age` and this relational operator only works for strings and integers. Equality will work in a much broader context: objects, symbols and most other types are acceptable to the '=' operator.

Find all animals which are either 10 years or older, or that are tigers:

```
(find-all *prot*
  :return
  '(
    (the-animal animal)
    )
  :where
  '(or
    (>= (slot-value the-animal age) 10)
    (= (slot-value the-animal species) tiger))
  )
```

Returns: ((<*tiger*>) (<*camel*>) (<*tiger*>))

Note the duplicate in this result. This is due to the fact that <*tiger*> qualifies once because it is 10 years old, and once because it is a tiger.

Find all animals that are cared for by anybody:

```
(find-all *prot*
  :return
  '(
    (the-animal animal)
    )
  :with
  '(
    (care-link cares-for-link)
    )
  :where
  '(= the-animal (slot-value care-link right))
  )
```

Returns: ((<*tiger*>) (<*camel*>))

Go through all the `cares-for-links`. Filter out the ones where the animal being cared for is older than the ranger caring for it. Then, for each `cares-for-link` that qualifies, return the ranger (the link's `left` slot), the animal (the link's `right` slot) and the number of hours per week the ranger spends caring for the animal (the link's `hours-per-week` slot):

```
(find-all *prot*
  :return
  '(
    (the-ranger  ranger)
    (the-animal  animal)
    (the-hours   integer)
    )
  :with
  '(
    (a-link cares-for-link)
    )
  :where
  '(and
    (= the-ranger (slot-value a-link left))
    (= the-animal (slot-value a-link right))
    (= the-hours  (slot-value a-link hours-per-week))
    (> (slot-value the-animal age) (slot-value the-ranger age))
    )
  )
```

Returns: ((<*mary*> <*camel*> 30))

Queries may use built-in or user-defined database functions also. This means that if one is familiar with the internals of the database, one may use this knowledge even at this level of abstraction. Find the names of all user types (`UserTypeNm` is one of Iris' built-in functions):

```
(find-all *prot*
  :return
  '(
    (name simple-string)
    )
  :where
  '(= name (iris::UserTypeNm))
  )
```

Returns:

```
(
 ("RANGER"                          ")
 ("ANIMAL                           ")
 ("CARES-FOR-LINK                   ")
                 .
```

```
                         .
                         .
 )
```

Find all animals:

```
(find-all *prot*
  :return
  '(
    (the-animal  animal)
    )
  :where
  T
  )
```

Returns: ((<*tiger*>) (<*camel*>))

12.2 Long Information in Slots

This section describes a PCLOS problem that occurs only for Iris databases. A clean fix does not currently exist for it, but it is very low-level and can be skipped on a first reading.

The Iris database has a potential problem when the sum of the lengths of the database representations of all slots of an object exceeds 4096. PCLOS attempts to take care of this by storing particularly large slot values separately. The "threshold" used is stored in `pclos:*field-length-limit*`. Whenever the database representation of a slot value exceeds that number, it is stored in a separate file with proper back-pointers into the database. References to the long information in the long-information file are placed in the database where the long information would normally be stored. This works well in all cases but when enough slots are just below that threshold to cause the overflow. In that case PCLOS signals an error. By changing the threshold the programmer can then avoid the problem.

Apart from the fact that it is not bullet-proof, this mechanism has two disadvantages: PCLOS must be able to open the long-information file to store long information in and the pointers between the long-information file and the database must be maintained when a short value is placed into a slot which previously contained long information.

The second problem is "solved" by a garbage collection routine which may be run on a database at any time. The references are not cleaned up while values are set because this would be prohibitively expensive. Not doing the garbage collection causes the long-information file to grow unnecessarily, but has no impact on the size or integrity of the remainder of the database.

The garbage collection routine is invoked by:

```
(pclos::long-string-gc (database <protector>))
```

Opening the long-information file may become a problem when Iris is used in server mode. In that case there must be a remote file access (RFA) path to the remote server into the directory where the database resides. This access must allow reading and writing. PCLOS will attempt to open the long-information file using the database user name, the name "guest" and the name "rfa" in turn. If all these attempts fail, PCLOS will signal a continuable error. If the user continues, operation proceeds without the long-information "fix". If any long information is subsequently encountered, an error is signalled. The PCLOS user may avoid the continuable error by setting the variable `*ignore-long-information-problems*` to `T`. In that case the system behaves as if the continuable error had been issued and continued.

13 The In-Core Database

This database has the obvious disadvantages of single-user, in-memory data stores.

13.1 Archiving

In the context of the `find-protector` command it was explained how one may cause a new in-core database to load information from an archive that was previously stored there by another in-core database. Two operations are available to cause an in-core database to save all its state to an archive and to restore from one:

```
(save-to-archive <protector>
                 &optional
                 (archive-name NIL))
```

If `archive-name` is provided, it must be a file path. The method will write all of the database's state to the specified archive. In this case the path must be writable in the Unix sense. If `archive-name` is not provided, the database's name is used. Recall that this is the second element in the list passed to `find-protector` when the database is made.

```
(restore-from-archive <protector>
                      &optional
                      (archive-name NIL))
```

This will load all information stored in the archive into the database. Note that no application objects will be internalized or created in any other way as part of this operation. It will succeed even if some classes stored in the database are not defined in the current session.

Note that this command will destroy any information that is currently stored in the database. It will also invalidate all internalized objects.

If `archive-name` is provided, this will read the archive which must have been produced by a `save-to-archive` of some in-core database. The state contained in the archive will then be the new contents of the database. The name must be the path to a readable file in the Unix sense. If `archive-name` is not provided or is NIL, the database's name is used. Recall that this is the second element in the list passed to `find-protector` when the database is made.

`Restore-from-archive` currently has an unpleasant limitation: all Lisp packages of symbols in the archive must exist in the current Lisp session, and all symbols that were exported from these packages at the time the archive was created must have been exported in the current session also. If these conditions are not met, an error will occur. The problem can then be fixed and the command may be reissued.

14 Some Examples

Here are a few samples of how a programmer operates in PCLOS.

```
;; Load the core of the system:
(require "pclos")

;; Now get the interface to a particular session of a
;; particular kind of database:

(setf *protector*
  (find-protector *master-protector*
                  '(:iris "smith@hpldbserv:/dbtests/testdb")))

(defclass TEST ()
  (
   (slot0 :initform 10)
   (slot1 :initform 20)
   )
  (:metaclass pclos-class)
  )

;; Create a transient object:
(setf obj0 (make-instance 'test))

;; Make the object persistent:
(protect obj0 *protector*)

;; Now handling of the 'test class and of obj0 are not different from
;; ordinary Clos. For example: Set a slot (will modify database directly):
(setf (slot-value obj0 'slot0) 100)

;; User-Controlled Caching and Transactions:

;; Define checkpoint to roll back to:
(begin-transaction *protector*)

(setf (slot-value obj0 'slot0) 200)
(setf (slot-value obj0 'slot1) 300)

;; Abort everything done since beginning of transaction:
(abort-transaction *protector*)

;; Now slot0 is 100 and slot1 is 20.

;; Cache the object under the protection of a transaction
;; to make access to its slots faster:
(begin-transaction *protector*)
(cache obj0)
```

```
;; Modify in-memory copy of a slot. Nobody can write to database
;; copy, since Iris has concurrency control:
(setf (slot-value obj0 'slot0) 500)

;; The following abort will reset slot0 to 100 and will cause obj0
;; to be no longer cached:
(abort-transaction *protector*)

;; Cache object without transaction protection, i.e. take a snapshot:
(cache obj0)

;; The following transaction will not protect the object copy in the
;; database because obj0 is already cached:

(begin-transaction *protector*)

;; Modify in-memory copy of a slot. Database copy may meanwhile
;; be overwritten by others:
(setf (slot-value obj0 'slot0) 500)

;; The following abort will affect the in-memory state of obj0 only.
;; Modifications done by others to obj0 in the database are not rolled back.
;; Afterwards slot0 will contain 100:
(abort-transaction *protector*)

;; Update the in-memory snapshot of obj0 to reflect its state
;; as known by the database:

(recache obj0)

;; Finding objects:

;; Find one of any objects of class 'test in the database
;; represented by *protector* for which slot0 is equal to 50:
(setf obj1 (find-one *protector*
                     'test
                     '(= slot0 50)))

;; Find all instances of class 'test:
(setf instances (find-all *protector*
  'test
  t))
```

15 Limitations

The following sections summarize current PCLOS limitations the programmer should be aware of.

15.1 Global Limitations

The following limitations are present for all databases:

Execution Speed When running with transient or cached objects, execution speed is comparable to execution speed with standard Clos but the benefits of using `with-slots` for compile-time optimization is lost. This does mean some loss in execution speed. All slot accesses will be equivalent to using `slot-value` in PCLOS classes. The reason for this is simply that the values of slots might be in the database or in memory, depending on whether the respective object is transient, cached or persistent and uncached at run-time. No compile-time optimization is therefore possible.

Inheritance from non-persistent classes: This is legal, but when methods of the non-persistent parent class are invoked with an instance of the child persistent class, slots will be accessed incorrectly if `(with-slots ((:use-accessors NIL)) ...)` is used. All is well if `:use-accessors` is T, or if slot-value is used to access slots in methods of the non-persistent parent class. This can probably be fixed when PCLOS is ported to some later version of the Clos implementation when method deoptimization is supported.

It is illegal for a non-persistent class to inherit from a persistent class.

Redefining persistent classes: Class redefinition should ideally modify all class information and instances in all databases and in memory. PCLOS does not currently do this. Class redefinition is possible with the understanding that the database will not be modified as part of the redefinition. This means that any attempt to protect instances of the newly shaped class will cause an error if the previous shape of the class was known to the database, that is if the class has been protected before. The `unprotect` operation on the class or, more brutally, the `destroy-class` operation on the protector may be used to remove all knowledge of the class from a database. In order to remind the programmer of this limitation, redefining persistent classes will signal a continuable error in a standard PCLOS environment. This can become annoying during debug sessions, and the error signalling can be disabled by setting the variable `pclos:*allow-persistent-class-redefinition*` to T.

It is always illegal to redefine classes while in a transaction.

Recursive class definition: It is currently not possible to type slots of a PCLOS class to another PCLOS class which in turn contains a slot typed to the first class. Example:

```
(defclass man ()
  (
   (partner :type woman)
  )
  (:metaclass pclos-class)
```

```
  )

(defclass woman ()
  (
   (partner :type man)
  )
  (:metaclass pclos-class)
  )
```

This will signal a clearly self-describing error as soon as an attempt is made to protect any instance of class `man` or `woman`. The problem is not fundamental and can be fixed. Other than type checking and cleanliness nothing is lost by not typing slots when the type is another PCLOS class. Note, however, that for some complex query operations typing is a good idea for types other than PCLOS classes because operators like '>' are acceptable only if slots are correctly typed (see section on Iris complex queries).

Slots typed to `symbols`: When typing a slot to be `symbol`, setting the slot to `NIL` may cause a problem: `NIL` and `T` are mapped by PCLOS to type `boolean` on all databases that support such a type. Lisp considers `NIL` to be a `symbol` and therefore allows its assignment to slots typed to contain `symbols`. But when a database is asked to put value `NIL` into a slot that was typed to contain `symbols`, the database will complain that a `boolean` is assigned to a slot that requires a `symbol`.

***Eq*ness:** Lisp *eq*'ness is guaranteed for symbols and objects for regular operation and for transaction rollback. But items of all other Lisp types are not guaranteed in this respect. Example:

```
(setf (x obj1) '(a b c))
(setf (x obj2) (x obj1))

(eq (x obj1) (x obj2)) => NIL
```

which is not what standard Clos would do. Reason: The two slots really *are* different items, namely strangely encoded bits in a database. In cached objects one might actually observe *eq*ness, but it is a bad idea to rely on this coincidence. In particular, after a transaction rollback, *eq*ness is not going to be valid for cached objects either.

Substructures of slots: This one is unpleasant: One cannot set substructures of slots. Example:

```
(setf (x obj1) '(a b c))     ; fine
(setf (second (x obj1)) 'z)  ; LOOKS fine, but
(x obj1) => (a b c)
```

Instead one has to write:

```
(setf foo (x obj1))
(setf (second foo) 'z)
(setf (x obj1) foo)
```

Reason: The setf method of **second** knows nothing about persistent slots. This could be fixed for all `setf` methods one could think of, for instance `first` through `tenth` and `svref` and `elt` and .... but there will always be one more, and a way would have to be found to ensure proper behavior for `setf` methods defined by the programmer.

One mode of operation that has worked is to assign slots to local (`let`) variables, to have the method code operate on these local variables, and to assign the locals back to the corresponding slots at the end of the method. A more luxurious way would be to write a macro analogous to `with-slots` that would accomplish this automatically.

Legal datatypes: As described in the section on type conversion customization, most Lisp datatypes are supported on all supported databases. But this is not true for multi-dimensional arrays and array displacement. Please see the above-mentioned section on type conversion customization for ways to provide mappings for these types also.

Required compile script fix: Due to the fact that the Hewlett-Packard CommonLisp compiler is not reentrant, it must be ensured that the PCLOS metaclass is known when files are compiled that define PCLOS classes. If this is not done, the compilation will still work, but it will take much longer.

This problem is taken care of by including the following two lines in the compilation script (which is often called `clfaslfile`):

```
(require "pclos-load")
(require "pclos-meta")
```

The best place is just before the `(compile-file ...)` entry of the script.

Searching through complex structures: Searching will only find items which are stored at the "top level" of slots. It is not possible to search for items that are, for instance, enclosed in lists or vectors.

15.2 Limitations Specific to the In-core Database

The following limitations are only relevant if the In-core database is used with PCLOS:

Packages required for database restoration: When an In-core database is restored from an archive, all Lisp packages of symbols in the archive must exist in the current Lisp session, and all symbols that were exported from these packages a the time the archive was created must have been exported in the current session also. If these conditions are not met, an error will occur. The problem can then be fixed and the command may be reissued.

Part II

PCLOS Version 2.1

16 Introduction

This part describes the differences between PCLOS Version 2.0 and Version 2.1. The major differences are that PCLOS 2.1 is compatible with a newer version of Iris, that a third database has been added and that one PCLOS call was changed because its semantics confused users.

17 Random Improvements

17.1 The `find-protector` Optional Argument

The call to `find-protector` takes one optional parameter which has different meanings for the various databases. For the Iris database that parameter indicates the size of the log file, which is set to a reasonable value if the parameter is not provided in the call. The meaning for the in-core database is the one that has been changed for this release. It used to allow the programmer to trigger an automatic restoration from archive of the in-core database about to be accessed. This seemed to confuse users. The optional parameter has therefore been deactivated for in-core databases. This means that the first call to `find-protector` with the in-core specification will always return a protector to an empty in-core database. If this database is to be charged from some archive, an explicit `restore-from-archive` message must subsequently be sent to the protector.

18 The Workspace Database

The Workspace is an in-memory database which is maintained in a separate server process from the PCLOS client - possibly on a different machine. This database is not included in the public PCLOS system, but the interface to it is provided for reference. It is accessible to multiple clients simultaneously, but **no** concurrency control is currently provided. Like the single-user in-core database available since PCLOS 2.0, Workspace databases may be archived to a file and retrieved later on. Since the Workspace database is in-memory, it is faster than the Iris database and PCLOS caching might not be necessary as often.

The Workspace offers additional capabilities beyond data storage. These include in particular the ability to register methods and to use the server as a message dispatch hub. None of these capabilities have been introduced into the PCLOS model. For details about them, see Brian Beach.

Note that currently only one Workspace database may be open at any time.

18.1 PCLOS Access to the Workspace

Access to the Workspace from PCLOS is identical to access using Iris or the in-core database with the exception of any special limitations noted in section 18.2. Applications should run on any of these three PCLOS databases as long as core-level operations are used.

The `find-protector` message now recognizes the additional database specification `:workspace`. Example:

```
(setf prot (find-protector *master-protector*
                           '(:workspace ''/databases/testdb'')))
```

This call will return a protector which will use a Workspace server process for its database. The meaning of the optional third parameter to `find-protector` is a cache size. The Workspace database implements its own internal caching which is **independent** from the PCLOS caching mechanism. This cache size is the number of objects that are to be kept in cache and it is set to a reasonable default if the optional parameter is omitted or has incorrect contents: it must be an integer greater than 0.

18.2 Limitations of Workspace Databases

The following limitations are currently known:

- Integers n stored in the workspace must satisfy $-2^{31} < n < 2^{31} - 1$.
- Information is currently stored in the workspace in a slow and space-wasting way whenever items of non-native types are involved. This does not otherwise affect PCLOS users. When the problem is fixed, no application changes will be necessary.
- Unprotecting a class cannot currently be rolled back.
- Only one Workspace process is usable at any given time. This is enforced by PCLOS.

Part III

PCLOS Version 3.0

19 Introduction

The PCLOS 3.0 release contains changes done to allow upgrading to new subsystems, and to add or correct functionality.

Here is a list of the major differences between the PCLOS versions 2.1 and 3.0:

- Support for the Workspace object base was updated to be compatible with the March 1989 version of the Workspace software. Object-oriented access to demons and the IPC capabilities of the Workspace has been provided. There is no longer a size limitation for integers, and rollback of `unprotect` on classes now works.
- A new *write-through* mode of operation was added to avoid problems encountered in the past with *uncached* operation. Full backward compatibility has been preserved.
- The ability of PCLOS to keep track of whether objects were modified and need updating has been worked over and made quite reliable even in the face of transaction rollback. Class objects now also support the `obj-dirty?` message.
- The PCLOS code was modified to adhere to a configuration configuration scheme and versioning conventions.
- In the PCLOS 2.1 release PCLOS did not consider packages in its comparison of class names. This meant that no two classes could have the same print name, even if they were in different packages. This has been fixed.
- Performance related modifications:
  - Rollback support for cached information has been made significantly more time and space efficient.
  - The code was inspected and modified to allow compilation with less safe, but faster optimizations in key parts of the code.
  - Slot access mechanisms were modified to improve performance for slot accesses to both unprotected and protected instances.
  - The Workspace interface is now more space and time efficient for all data structures other than integers and object references.
- Transient slots are no longer followed in the recursive functions like `recursive-cache`. For programmers using the `apply-to-object` recursive apply facility, an option has been provided to control whether transient slots should be considered in the recursive walk.
- A recursive object lattice walker has been made available.
- A fully automatic, object-oriented test suite for PCLOS is now available to allow convenient regression testing after code modifications.

In the remainder of the document each change or addition is described in turn. Some of the sections clarify issues that were omitted or whose description was not clear enough in earlier documentation.

20 Startup Procedure

In order to load and run your PCLOS you need to load the PCLOS *configuration* file. This file is delivered to you in *<pclos-root>/admin/pclos-config.l*. This default copy is correct for the average system. As in the installation procedure, the configuration uses the *$pclos* environment variable if it is defined. Otherwise the directory */users/pclos* will be used. As in previous releases, you may set the global Lisp variable **pclos** to override this default behavior. This is documented in the original PCLOS 2.0 Manual.

To load your configuration and PCLOS, evaluate the following expression in your Lisp environment:

```
(config:register-and-require-config "pclos"
                                     "<pclos-root>/admin/pclos-config.l")
(require ``pclos'')
```

After PCLOS is loaded, you may obtain version information by evaluating the expression:

```
(config:find-app-version "pclos")
```

21 Upgraded Workspace Support

The two major Workspace changes that are relevant to object persistence are the addition of concurrency control primitives and support for rollback. Changes of PCLOS's use of the new Workspace software should be transparent to programmers using PCLOS, except for the support of rollback for the `unprotect` operation on class objects. PCLOS 2.1 made this capability available for the Iris and In-core object base, but not for the Workspace. This capability is now available. Here are details on changes or additional functionality.

21.1 Large Integers

In the 2.1 release, integers stored in the Workspace were required to be in the range: $-2^{31} < n < 2^{31} - 1$. This limitation no longer exists.

21.2 Access to Demons, IPC and Method Operations

The latest Workspace software provides new capabilities in the areas of demons and interprocess communication. These capabilities have been made available to the users of PCLOS. They are not part of the official PCLOS release and have therefore not been incorporated into the PCLOS system model. Instead, programmers may get access to the CLOS object that represents the interface to the Workspace. The relevant Workspace operations are then controlled through messages to this Workspace object.

The Workspace interface object is obtained from the *protector* associated with the particular Workspace session:

```
(workspace-db (database <protector>))
```

Here is a list of methods that are supported. Please consult the Workspace documentation for details on how to use these methods. Note that you need to be careful not to interfer with PCLOS' use of the Workspace interface object. In particular, you should use the PCLOS `begin-transaction`, `end-transaction` and `commit-transaction` operations, instead of directly using the respective Workspace interface methods below. The use of the other primitives is less critical, because their functionality is not part of the PCLOS model. Interference is therefore less likely. Do remember that access to these low-level methods is not part of the PCLOS release. It is not subject to the same degree of testing that is done on the release as a whole.

Messages:

```
ws-handle-messages
ws-handle-message
ws-next-lisp-message
ws-send-answer
ws-await-messages
ws-messages-pending
```

Transactions:

```
ws-transaction-begin
ws-transaction-lock
ws-transaction-abort
ws-transaction-commit
ws-transaction-add-object
ws-lock
ws-commit
ws-abort
```

Demons:

```
ws-prim-set-demon
ws-set-object-demon
ws-remove-demon
ws-handle-demons
```

Methods:

```
ws-add-method
ws-remove-method
ws-call
ws-handle-method-message
ws-method-names
```

Misc:

```
ws-find-type
ws-socket-id
ws-client-id
```

21.3 Queries In Open Transactions

When issuing queries to the Workspace through PCLOS, you should keep in mind that modifications to Workspace data is only "seen" by queries after the changes have been committed. When you therefore begin a transaction and make changes to objects, any queries by you or by others will not "see" these changes. They are made visible when you end the transaction.

21.4 Workspace Query Anomaly

While the visibility rules described above are correct behavior, the following query aspect of PCLOS on the Workspace is incorrect. Since neither Lisp nor the Workspace are typed, a given slot may contain differently typed values in different instances of a class. Datatypes not native to the standard Workspace are represented with some appropriate encoding. This means, for example, that a query requesting all instances of some class whose slot `foo` is greater than 10 might return instances whose `foo` slot contains, for instance, a Lisp symbol. The Workspace query processor cannot distinguish between slot values that contain a user's integer and values that contain some encoded information that happens to look like an integer. If type discipline for slot values is maintained by programmers, this is obviously not a problem. But since values of different types are legal in CLOS slots, this behavior can be confusing.

This same problem would exist on the Iris interface. Iris, however, has stricter type restrictions than the Workspace when applying functions as parts of queries: Error conditions are, for instance, signalled when a comparison operator is applied with one string and another integer argument.

22 The New *Write-Through* Mode

This section describes a new mode for objects and their classes. Related modes that have been available in previous releases include for example: *cached*, *protected* and others. The new *write-through* mode was added to fix problems that occurred in the *uncached* mode.

The following subsection explains why this mode was introduced. Then the technical details are presented.

22.1 Why This New Mode?

A proliferation of system modes tends to be confusing. But the decision on eliminating modes must be based on experience. An initial system must be flexible, because the correct behavior and preferred usage is not yet clear. Once the preferred way of operating has been determined, options may be eliminated safely, or may be consolidated into option packages.

In this case a mode has been added to avoid problems that have been observed when protected objects are not *cache*d. *Uncached* operation is attractive when object slot updates are to be propagated to the object bases as quickly as possible, and when slot reading is to produce the freshest values possible.

Without caching, each reading of a slot will bring the slot's value into memory from the object base. Writing of slots is not done in memory, but is immediately pushed into the underlying object base.

A problem arises with slot reading in *uncached* operation. Space for the slot value being retrieved is allocated each time the slot is read. The resulting performance cost is obviously high if the persistent slots are accessed frequently. For infrequently accessed slots, or for fast object servers, this cost can be acceptable. The problem, however, lies in the fact that if a program reads a slot twice in a row and the value is not an object or a symbol, the two results will have the correct contents, but will be in different locations. Since Lisp often uses `eq`ness, this can lead to problems. This difficulty has been documented in the section on PCLOS shortcomings in the original PCLOS 2.0 Manual.

Another problem of *uncached* operation was also documented in the 2.0 Manual: it arises when destructive operations are performed on vectors or lists. These operations do not cause a slot to be updated. Instead, they simply modify information the slot points to. In *uncached* mode these changes were lost. A workaround has been the introduction of local variables which are initialized to slot values, and which are assigned to the respective slots at the end of methods.

A new mode is therefore needed that avoids these problems, but retains as much as possible of the object base immediacy of the *uncached* mode of operation.

22.2 Definition of the *Write-Through* Mode

Both objects and classes may be put into and out of `write-through` mode. This is done through the operations:

```
(write-through <instance-or-class-object> &key
                                          (affect-class-slots T)
                                          (preserve-cached-values T))

(recursive-write-through <instance> &key
                                    (affect-class-slots T)
                                    (preserve-cached-values T))

(unwrite-through <instance-or-class-object> &key
                                            (affect-class-slots T))

(recursive-unwrite-through <instance> &key
                                      (affect-class-slots T))

(write-through? <instance-or-class-object>)
```

In the *write-through* mode every slot *update* will be done both in memory and in the object base. This ensures that the new value is propagated as quickly as possible. When slots are *read*, the value from memory is used, just like in the *cached* mode. This avoids both the `eq`ness problem and the difficulties with destructive operations. It does mean that information in slots may be stale, unless the programmer takes precautions. These may use one of the following mechanisms:

- The use of transactions to prevent the slots from being written by others.
- Client-managed locking.
- Properly placed `recache` operations.

You can think of the *write-through* mode as causing the same behavior as the *cached* mode, with the exception that slot updates are written to the object base. It is indeed true that an `unwrite-through` operation on an object that has been in *write-through* mode will transfer the object into the *cached* mode.

The keyword arguments have the same meaning they have for the `cache` and `uncache` operations: The `:preserve-cached-values` argument controls whether those slots that have been cached separately should be overwritten with the respective values from the object base when the object is put into *write-through* mode; remember that a transition to the *write-through* mode will have to bring the slot values from the object base into memory just like the `cache` operation does.

The `:affect-class-slots` argument controls whether the operation should automatically be done to the object's class. The default of this argument is `T`. This means that any class-allocated slots will track the state of the object involved.

Unless the programmer uses the `unwrite-through-all` operation (documented below), PCLOS will cause all persistent objects to be in *write-through* mode by default! The `unwrite-through` operation may, of course, be used at any time. The operations `uncache` and `abort-cache` will also take an object out of *write-through* mode. The `describe` operation on objects will show whether an object is in *write-through* mode.

The capability of *write-through* mode is not limited to instances. Class objects may also be put into *write-through* mode. This means that all class-allocated slots are treated in the *write-through* way. This may be useful for applications that want to treat the slots that are private to each instance differently than the slots that are shared among all instances. Here are two example scenarios: many instance-allocated slots may be kept transient for maximum performance. The class-allocated slots, on the other hand, would be persistent and in `write-through` mode. This would extend the sharing intrinsic in class-allocation to the entire client community. It is indeed possible to define classes with some class-allocated slots, and to keep all instances of the class *unprotect*ed. The class, on the other hand, could be protected and the class-allocated slots could be used for sharing among instances of the class, both locally and throughout the client community.

The `write-through`, `unwrite-through`, `recursive-write-through` and `recursive-unwrite-through` operations will be rolled back when performed within a transaction that is subsequently aborted.

22.3 Definition of the *Write-Through-All* Mode

The *write-through-all* mode is an attribute of class objects only. It is set and reset by the following operations:

```
(write-through-all <class-object>)
(unwrite-through-all <class-object>)
```

When in *write-through-all* mode, a class will ensure that all its **new** instances will be in *write-through* mode. This includes both newly created instances and instances that are brought into the current environment through queries. Instances that are already in the environment when the `write-through-all` operation is performed will be unaffected.

Both the `write-through-all` and the `unwrite-through-all` operations will be rolled back when performed within a transaction that is subsequently aborted.

22.4 Backward Compatibility

The `cache`, `uncache` and `abort-cache` operations are still available. If an `unwrite-through` operation is applied to an object that is in *write-through* mode, the object will revert to the *cached* mode. If you do take objects into the *uncached* mode, do keep in mind the two problems explained above.

23 Performance-Related Modifications

Various changes have been made that address performance issues. While persistence is still expensive, some operations have been optimized. The following presents some details.

Since these implementation modifications introduce no semantic changes, this section may be skipped by anyone not interested in implementation details.

23.1 Modified Rollback Support

The support for rollback of modifications to objects that are *cached* or in *write-through* mode has been modified to scale up, and to minimize information copying.

The original strategy for supporting rollback for *cached* objects was to copy the objects at the beginning of a transaction. This places all of the cost of rollback support at the beginning of the transaction. This is desirable, but it will work only if relatively few objects are cached. The performance limitations even of relatively fast object servers forces caching to be used in most cases. PCLOS was therefore modified to use lazy copying which only saves information when it is to be updated. Once cached information has been modified once, subsequent accesses incur a minimal overhead.

23.2 Compiler Optimizations

The CommonLisp compiler is capable of optimizing code when instructed to do so. These optimizations cause assumptions to be made about the shape of data structures and about the correctness of function parameters. Code to be optimized needs to be written with these assumptions in mind, so that the resulting binaries operate correctly. PCLOS has been modified to allow optimizations at least at key sites of bottlenecks such as slot access.

23.3 Slot Access Optimizations

In addition to the compiler optimizations, the slot access methods have been hand-tailored to do as little work as possible. This should affect the performance of slot access to both protected and unprotected instances. Automatically created slot accessors are no longer generated. This enhances performance and saves space.

24 Transient Slots

PCLOS must often recursively work through an object lattice. This is done, for instance, when an object is `protected`. All objects reachable from the respective object will also be protected. Another example is the `recursive-write-through` operation. Beginning with this release, transient slots are no longer considered when traversing object lattices. This means on one hand that the programmer is free to put any information into transient slots without worrying about PCLOS getting confused or irritated. On the other hand, care must be taken to remember that objects reachable only through some transient slot will not be affected by all of these recursive operations.

25 Recursive Object Lattice Walker

The facility used by PCLOS to run through object lattices may be useful for other purposes. It has therefore been made available for use by PCLOS users. This is a system function which **trusts its users**. No argument or sanity checking is done! This function is for knowledgeable, careful, macho men and women only.

```
(apply-to-object (object client-function
                          more-args &key (test NIL)
                         (obj-first NIL)
                         (fetch-slots-first NIL)
                         (notify-if-circularity NIL)
                         (do-transient-slots T))
```

Given an `object` and a `client-function`, this routine goes through the object's slots. If it finds an object anywhere hidden in the slots, it `apply`'s function `client-function`, passing it the object and `more-args`. At the end it `applys` `client-function` to the object originally passed. The `more-args` parameter may be `NIL`. Otherwise it must be a list of arguments. The `apply-to-object` function returns a list of objects to which the `client-function` was applied.

If keyword argument `:obj-first` is `T`, the `client-function` call will first be done to the object itself and then to any objects that are reachable through its slots.

The keyword argument `:test`, when provided, should be a function expecting an object as the sole parameter. Whenever `apply-to-object` is about to apply `client-function` to an object, it will pass the object to the `:test` function. The call of the `client-function` will be done only if the `:test` function returns non-`NIL`.

If the key :notify-if-circularity is non-NIL, it must contain a function to be called when a reference circularity is detected. This function will be called when apply-to-object detects that one of the slots of some object points back to an object it already looked at. The function will be called with two arguments: the object that is being referred back to (i.e. the starting point of the cycle during the recursive walk), and a *cons* cell whose car is a reference count and whose cdr is T or NIL, depending on whether client-function has already been applied to the object. After the function specified in the :notify-if-circularity keyword argument has been called, the slots of the object that is being referred back to will, of course, not be walked again. If :notify-if-circularity is unspecified, the circularity resolution will proceed normally without calling any client function.

The keyword argument :fetch-slots-first determines when an object's slot values will be fetched to prepare the continuation of the walk. If it is non-NIL, the slot values are fetched immediately when an object is encountered. This is done before the calls to client-function and test. Otherwise, slot fetching is done at some convenient time, but it is guaranteed that it will be called *after* the call to client-function in the case of :obj-first being non-NIL. This can be used when the client-function will change the slots of the object or will make access to them slower. An example is the *recursive-uncache* operation: it is preferable to fetch the slots needed for the continuation of the walk before the objects are uncached, because slot access will be slower once the uncaching has been done.

The keyword argument :do-transient-slots controls whether the routine also walks through subtrees accessible only through transient slots.

The apply-to-object function will find objects "hidden" in *cons* cells, lists, vectors and structs.

Figures 2 and 3 show an example of how to use the walker.

Assume that root-obj and other-object are *cached* but not in *write-through* mode. A program has caused modifications on the local copy of other-object and you want to update the database as cheaply as possible to match the local copies of your objects. Let us assume further that we do not know which objects have been modified. This means that we will need to look at each of the objects reachable from root-obj – in this example only other-object – and we have to save out the ones that were changed. Figure 3 shows how the object base may be updated with minimal work.

This will run through all the reachable objects and will write them back if needed. The use of the keyword argument :affect-class-slots NIL is an efficiency trick: as described in the 2.0 Manual, the write-back method will normally write back the object in question **and** the class slots. Since we will potentially call write-back many times with different objects, it is silly to apply this operation to the class every time. We therefore prevent that and do it explicitly at the end.

26 Server-Independent Object Base Naming

Assigning names to object bases to identify them uniquely introduces a dilemma. On one hand, it would be nice if it were possible to create and maintain object bases of the same name on different servers. This, however, implies that a server name would have

```
(defclass FOO ()
  (
    (slot1 :initform <some-object-reference)
    (slot2 :initform "Some info")
    (slot3 :initform 20 :allocation :class)
    )
  (:metaclass pclos-class)
)

(setf root-obj (make-instance 'foo))
(setf other-object (make-instance 'foo))
(setf (slot-value root-obj 'slot1) other-object)
```

Figure 2: A Class With Class-Allocated Slots

```
(apply-to-object root-obj
                 #'write-back
                 (list :affect-class-slots NIL)
                 :test #'obj-dirty?)
(when (obj-dirty? (class-named 'foo))
  (write-back (class-named 'foo)))
```

Figure 3: Updating the Database With Minimal Work

to be an integral part of the object base name. This would in turn introduce a different restriction: PCLOS object slots may reference objects that reside in other object bases. These references must obviously contain the identifying names of the target object bases. Such references can potentially be stored in any number of persistent objects on any number of object bases.

If servers were an integral part of object base names, every object base would have to remain on the server on which it was created. This is because the references to its objects could otherwise not be resolved: the object base would not be found at the time of reference resolution.

PCLOS therefore requires that names of object bases containing objects that are referenced from objects in other object bases must be unique independent of which server an object base resides on. With this policy, an object base may be moved to a different server. At run time a protector can then be created for the object base. Whenever references to objects in this object base are resolved, this protector will be found in the run-time environment. It will know about the *current* location of the target object base.

27 Improved Dirtyness Management

PCLOS has always been quite careful to keep track of whether an object or its class-allocated slots have been made inconsistent with the object base by local slot updates. There have been two changes which are described below. In addition, a more detailed explanation of the mechanism is presented.

27.1 Description of Changes

The `obj-dirty?` expression used to be a macro. It has been changed to be a method. This has the advantage that it can be used whenever a function is called for as a parameter. See section 25 for an example.

The `obj-dirty?` method may now be used both for instances and for class objects. The latter can be used to find out whether the class-allocated slots are dirty.

27.2 Some Helpful Details

Every object or class returns `T` or `NIL` to the predicate:

```
(obj-dirty?  <instance-or-class-object>)
```

If an object was clean at the beginning of a transaction and its slots are modified within that transaction, the object will be dirty. But if the transaction is aborted, the object will again be clean.

An object is marked as dirty when one of its slots is written into while the object or some individual slot are *cached*, but not in *write-through* mode. An object will be marked clean by an `uncache`, `recache`, `abort-cache` or `write-back` operation.

This dirtyness management may be used to make an object base consistent with the in-memory state of modified objects with a minimal amount of object base traffic. This can be particularly useful in conjunction with the `apply-to-object` operation described earlier. Note, however, that PCLOS does **not** notify clients when the object base copies of objects are modified by *other* clients. This means that an object could look clean, even though someone else has modified the object base copy of the object. Dirtyness is managed for all activities that happen locally. Some mechanisms for maintaining global consistency have been listed earlier.

In the context of consistency management the methods `get-cached-objects` and `get-cached-classes` are useful to remember. These are both methods on protectors, and they return a list of objects or classes that are under the protector's control and are cached.

28 Test Suite

A fully automatic, object-oriented test suite has been developed for PCLOS regression testing. It performs more than 100 individual tests and does not require programmer attention. It allows the introduction of `:before` and `:after` function hooks which cause debugging functions to be called before or after individual tests. While this set of tests will by no means find all the bugs, it does help with preliminary regression testing. The details are described in a separate document.

29 Clearing An Environment

It is often useful to start over with a clean slate, but without having to leave and reenter a run-time environment. Achieving this in the context of PCLOS is a very tricky problem, and while this release has improved this aspect, cleanup of an arbitrarily messy environment is still one of PCLOS' weaker points. No new capabilities have been added for this release. But here is an attempt to give guidance for what to use when. It is based on practical experience. Since by definition, the respective environments are in more or less bizarre states, empirical evidence must, and often can suffice.

There are varying degrees of PCLOS resets:

- The local copies of objects or classes may be invalidated. The associated object base itself is not affected.
- A given protector may be reset. This will invalidate all the objects and classes managed by it, and it will reset the object base interface object. A subsequent `find-protector` operation will return the same protector. The associated object base itself is not affected.
- A given protector may be destroyed. This will invalidate all objects and classes managed by it, and will reset the interface to its associated object base. A subsequent `find-protector` operation will return a new protector object. The associated object base itself is not affected.
- An object base may be destroyed.

To *invalidate* an object or class means to cut its connection with the object base on which it is persistent. Slot values will be set to `*invalid-data*`, unless this is specifically inhibited in the call to the invalidating method (see below). In that case the slot values will in general be undefined. If your environment is still healthy enough to cache objects, you can do that and *then* invalidate these objects. This will make them look like regular unprotected objects with the proper slot values.

Individual objects or classes may be invalidated by:

```
(abandon <instance-or-class-object>
         &key (invalidate-slots T))

(recursive-abandon <object>
                   &key (invalidate-slots T))
```

This destroys everything the object's or class' protector and its object base interface know about the object's or class' in-memory copy. The object base itself remains untouched. For objects, this can be rolled back with a transaction abort. For classes it cannot be rolled back. An attempt to execute this method on a class while in a transaction will cause an error to be signaled.

```
(abandon-known-objects <protector>
                        &optional (invalidate-slots T))
```

This invalidates all objects known to the specified protector and the operation cannot be rolled back. This method signals an error when attempted while in a transaction.

```
(destroy-protector (<master-protector>
                    protector
            &optional (invalidate-slots T)))
```

This will **not** destroy the associated object base. But it will cause the master protector to forget that the given protector exists. The protector's cached information – and consequently the object base interface's cached information are destroyed. All objects and classes associated with the protector are invalidated.

```
(destroy-database <protector>
  &key (invalidate-slots T))
```

This will physically destroy the object base and will reset the protector. Do think before using this operation.

Experience has shown that the **destroy-protector** operation – possibly in conjunction with **abandon** on at least some of your classes will yield the best results. The **destroy-protector** operation will cause the protector to **abandon** all the classes it knows about. But sometimes the state of the environment can get messy enough for the protector to miss some classes.

30 Miscellaneous

30.1 Dualism Of Operations

It is worth a reminder that just about all of the PCLOS operations may be applied both to instances and to their classes. Operating on classes will affect all class-allocated slots, while the same operations affect the instance-allocated slots when applied to instances. Class objects may be **cached**, put into *write-through* mode, **recached**, written back, etc. Unprotecting a class will unprotect all instances and will remove all traces of the class' representation in the object base. This operation, like all the others, can be rolled back when executed within a transaction.

30.2 Class Names

In the PCLOS 2.1 release, symbol packages were not considered when comparing class names. This meant that no two classes could have the same print name, even if they were in different packages. This has been fixed.

30.3 New Method to Find Protectors

The **find-protector** operation will create a protector object and will open an object base if a protector matching the caller's specification does not exist yet. A new method allows a search for such a protector with the guarantee that no protector will be created:

```
(find-protector-if-present <master-protector>
                           db-type-and-name)
```

The value **NIL** is returned if no matching protector is found. The **db-type-and-name** parameter is the same as for the **find-protector** operation.

30.4 New Method On *Finders*

It is now safe to close a finder any number of times. The method:

`(open? <finder>)`

has been added to the *finder* class.

30.5 Little Hints and Errata

Operations like `protect`, which cause a lot of data traffic to and from an object base, may be sped up considerably when done in a transaction which is committed when the work has been done. The reason for this is that multiple slot accesses will be done in memory only, if the system is in a transaction. Writing the final result back to the object base at the end – when the transaction is committed – will minimize network data traffic.

When `unprotect`ing a set of objects it is safer if you can use `recursive-unprotect` or the `unprotect` method on a class object. The reason is that these operations will resolve references among the objects they unprotect. When done one at a time, you may end up trying to unprotect an object which has a reference to a no-longer persistent object. When the object's slot values are pulled out of the object base as part of the unprotect, there will be an error because that reference cannot be resolved. The `recursive-unprotect` and `unprotect` on classes will try to do the unprotecting in a safe sequence.

Correction in the PCLOS 2.0 Manual: The index of the document should list page 14 as relevant to the `after-retrieve` method.

31 Conclusion

PCLOS now contains a large amount of functionality. It has grown in response to customer experiences and feedback. It is clear beyond doubt that object persistence is not an easy problem. I once read about a technique for building inner-city parks: the approach is to construct the park without pathways. After a while, the grass will be gone along the trails that are used by most people. The paved pathways are then added along those trails. PCLOS is in a somewhat similar state in that it contains many concepts and operations. Usage alone can show which of these are superfluous or plain wrong. What is then left will hopefully be of use.

32 Acknowledgments

Mike Creech has developed the conventions that unify the procedures for configuring, loading and distributing all of the modules in the system PCLOS was originally integrated in. In addition, he has provided many test cases and has made suggestions that have significantly shaped this PCLOS 3.0 release.

Bob Leichner, Vicki O'Day and Shari Jackson have helped to find many problems that arose when multiple machines shared data through the Workspace.

Brian Beach has provided many fixes and improvements of his Workspace server. Many new capabilities would not be available without his continued support.

Thanks also to Vicki O'Day for reading and improving the first draft of this document.

33 Appendix: Methods and Globals

allow-persistent-class-redefinition
database-trace
field-length-limit
ignore-long-information-problems
master-database
master-protector
pclos
abandon
abandon-known-objects
abort-cache
abort-transaction
add-adapter-missing-action
add-adapter-missing-action
add-customer-db-to-prog-converter
add-customer-prog-to-db-converter
add-database-native-type
add-facet
add-facet-missing-action
add-protocol-adapter
after-protect
after-retrieve
all
all-facets
all-operations
all-protectors
all-protocol-adapters
apply-to-object
associated-operations
before-protect
begin-transaction
cache
cache-slot
cached?
close-all-databases
close-all-finders
close-all-finders-by-class
close-database
close-finder
database-active?
database-name
database-type
describe
destroy-all-classes
destroy-class
destroy-database
destroy-protector
end-transaction
explain-all
explain-all-facets
explain-all-protocol-adapters
explain-facet
explain-protocol-adapter
explain-yourself
find-adapter-missing-action
find-adapter-missing-action
find-all
find-customer-db-to-prog-converter
find-customer-prog-to-db-converter
find-facet
find-facet-missing-action
find-facets-by-operation
find-one
find-protector
find-protector-if
find-protector-if-present
find-protocol-adapter
finder
get-cached-classes
get-cached-objects
in-transaction?
load-database
lock
make-protected-instance
more?
next
obj-dirty?
open-database
open?
protect
protected?
protector
recache
recache-slot
recursive-abandon
recursive-abort
recursive-abort-cache
recursive-cache
recursive-recache
recursive-uncache
recursive-unprotect
recursive-unwrite-through
recursive-write-back
recursive-write-through
remove-adapter-missing-action
remove-adapter-missing-action
remove-customer-db-to-prog-converter
remove-customer-prog-to-db-converter

```
remove-database-native-type
remove-facet
remove-facet-missing-action
remove-protocol-adapter
restore-from-archive
save-to-archive
slot-cached?
slot-values
uncache
uncache-slot
unlock
unprotect
unprotect-all-data
unprotect-instances
unwrite-through
unwrite-through-all
write-back
write-through
write-through-all
write-through?
```

Index

34 References

[1] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, Springer Verlag, 1988.

[2] Andreas Paepcke. PCLOS: A Critical Review. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[3] Andreas Paepcke. PCLOS: Stress Testing CLOS - Experiencing the Metaobject Protocol. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1990.

[4] Andreas Paepcke. User-level language crafting – introducing the CLOS metaobject protocol. Technical Report HPL-91-169, Hewlett-Packard Laboratories, 1991.

[5] D. Fishman et al. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, April 1987.

[6] Brian Beach and James Kempf. Doom: Permanent objects for common lisp. Technical Report STL-TM-86-09, HP Labs, September 1986.

[7] James Kempf, Andreas Paepcke, Brian Beach, Joseph Mohan, Brom Mahbod, and Alan Snyder. Language Level Persistence for an Object-Oriented Programming Platform. Technical Report STL-87-05, Hewlett-Packard Labs, October 1987.