Syntax and Semantics of a Persistent Common Lisp*

J. H. Jacobs and M. R. Swanson†
Center for Software Science
Department of Computer Science
University of Utah

Abstract

The syntax and semantics for UCL+P, a persistent Common Lisp, are defined. The definition provides adequate support for persistence while maintaining the look-and-feel of Common Lisp. All Lisp data types (except streams) can be made persistent. Persistence is conferred automatically on non-symbol values when they become part of a persistent data structure. Symbols are persistent if interned in a persistent package. The Common Lisp package facility was enhanced to allow for persistent packages which provide modularity to the space of persistent values and serve as the ultimate roots of the persistence conferral algorithm. Values are retrieved from the store using demand loading; new or mutated values are automatically detected and written back to the store when the transaction is committed. The sharing semantics of Lisp are preserved in this specification.

1 Introduction

The ability to construct programs that use persistent data (where values persist after a program execution) is becoming increasingly important in this era of gigabyte secondary stores. While database management systems adequately serve many business applications, they are insufficient to efficiently support many of the complex data structures required for modern computer applications such as CAD. Persistent languages are designed to serve the data management needs of these applications. Most persistent languages were evolved from existing languages such as: Algol [4], C++ [10] [15] [1], Smalltalk [8], ML [13], and Lisp [14] [11] [2] [3] [6]. In all but the Lisp languages the introduction of persistence has been fairly successful: the resulting language still has the look-and-feel of the original. Unfortunately the Lisp implementations have not been as successful in this regard.

Introducing persistence into a known language is a challenging undertaking, and when the original language provides automatic storage management the task is even harder. Seamless integration of persistence requires a thoughtful design, and careful, low-level modifications to both the compiler and runtime systems of the language. For C++ and the others, there was no choice but to bite the bullet and make the necessary low-level changes. However, the expressive power of Lisp provided shortcuts which attracted many implementors of persistence: restricting persistence to CLOS objects, limiting persistence support to interpreted code, etc. While these expediencies can produce usable systems, they are pale realizations of Persistent Lisp.

The UCL+P project began with the fundamentals. We first conceived of how a Persistent Lisp ought to work and look, then we undertook the necessary steps to make our concept a reality. In our view, a Persistent Lisp should follow three principles. First, it should conform to established Lisp syntax, semantics, and programming style; if it does not look like Lisp, it cannot be called Lisp. Second, the persistence features provided should be powerful enough so that the programmer need not resort to using Lisp I/O features or operating systems calls to implement persistent programs. Third, the programs constructed using Persistent Lisp should be sufficiently efficient that programmers will be able to use the programs that they construct. Following these three guidelines we designed and implemented UCL+P (Utah Common Lisp plus Persistence) by modifying the existing UCL compiler and runtime system. UCL+P features orthogonal persistence, concurrent transactions, and compiled code support. These features are integrated almost transparently into Common Lisp and, as a result, Lisp programmers will find that writing persistent programs is nearly as easy as writing volatile programs.

UCL+P forced us to address many interesting issues: low-level implementation support; language syntax and semantics; and persistent store design and interface. In this article we focus on the issues of language syntax and semantics. Creating UCL+P involved extending both the syntax and semantics of Common Lisp. We defined constructs to support atomic transactions and modular persistence. In addition, we had to define persistence semantics for all of the built-in data types provided by Common Lisp. Defining semantics for persistence required blending the requirements of shared, long-lived data with the semantics already defined by Common Lisp for volatile values to produce a coherent set of semantics. We believe that the semantics embodied in UCL+P fully support the manipulation of persistent values by Lisp programs and dovetail well with Common Lisp semantics.

The body of this article is grouped into five sections.

*This research was sponsored by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046.

†{jhjacobs,swanson}@cs.utah.edu

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

First, we provide an overview of the UCL+P system to help place the more detailed discussions in context. This is followed by a description of the syntax and semantics of the new transaction constructs. Continuing we look at how the `package` construct was extended to provide modularity for persistent values. Next we examine the general semantics of persistence, and finally turn to the persistent semantics peculiar to the specific Lisp data types.

2 An Overview of UCL+P

While the subject of this article is the language definition of a persistent Lisp, it may be helpful to have a brief overview of the complete system. UCL+P integrates persistence into Lisp. A transaction construct is provided to allow for atomic and concurrent access to persistent values (see section 3). UCL+P implements transactions using a commit-time validation concurrent transaction protocol which imposes little overhead when only a single program is accessing the store while providing full serializability when shared concurrent access to the store occurs. Note, however, that this implementation decision does not impact the language definition; a conservative locking approach could also be used.

Persistent values reside in modules called persistent packages, which are an extension of the Common Lisp package construct (see section 4). A UCL+P program may access any number of persistent packages simultaneously, and many programs can concurrently share a persistent package, subject to transaction semantics. Persistent packages may be arbitrarily large and may reside on one or more secondary storage devices managed by a store manager process. All access to the store is through the store manager. The store manager provides for atomic, multivalue updates to the store and insures that all programs using the store conform to transaction semantics.

Efficiently integrating persistence into Lisp required low-level modifications to both the compiler and runtime system of the Utah Common Lisp system (UCL)[9]. The compiler changes centered around two issues. An extra level of indirection was added to most intervalue references — thus pointers became handles (or double-pointers) — and an instruction was added to mark each referenced item as read or written when accessed. Replacing pointers with handles was necessary so that heap data which becomes persistent can be moved into the protected region of memory reserved for persistent values. Access to this region is blocked when outside the scope of a transaction. Value marking is necessary to implement both the transaction and automatic write-back mechanisms (see section 5.4).

After making these modifications we examined the size and performance of the resulting system. These modifications caused the code size of compiled applications to grow by 9%–16%, and changes to the runtime system caused the data area of programs to grow by 20%. The Gabriel benchmark set [7] was used to measure the effects of the changes on volatile-value-only performance. This is an important measure of system performance since most of the values used in a UCL+P program will be volatile and, once loaded, persistent values are accessed in the same way as volatile values.

The results showed that the amount of CPU time consumed by UCL+P versions of the benchmarks was a factor of 1.0 to 1.8 of that used by the corresponding UCL versions. The range is due to the different proportions of data types used in the individual benchmark; for example, list manipulation is significantly affected by the low-level changes made to the compiler, so list intensive benchmarks show a larger slowdown than others. Since a performance conscious programmer would favor the `vector` and `hash-table` data types over the `list`, the nominal slowdowns should be closer to the middle of the range. We have identified some future changes which should further improve the performance.

3 Transactions

Transactions ensure two related, but separate, properties for any actions that change the persistent values in the store. First, they provide atomicity for groups of related changes, ensuring the internal consistency to store values. Transactions group a set of operations on persistent values together and either makes the results of all the operations persistent or discards them all[5]. If the changes are made then the transaction is said to have *committed*, and otherwise the transaction is said to have *aborted*. Ensuring atomic transaction semantics requires that all mutations to persistent values occur only within the dynamic scope of a transaction construct.

Transactions also ensure the coherency of store and program data when multiple processes are performing concurrent accesses and updates to the store. Updates by separate processes are guaranteed to be serializable if they do not conflict; this may require aborting an update that would violate serializability. The coherency of the store is thus ensured. The coherency of *program* data is also important. That is, a program that simply reads persistent values but does not attempt to store any updates should still be able to depend on that set of values representing a single consistent view of the persistent store. Intervening updates could render the set of values inconsistent. Ensuring this consistent view of data for the program requires that all accesses to persistent data be within a transaction and that such accesses be recorded. Conflicting updates by other processes can then be caught by the runtime system and the transaction aborted.

Consequently, in UCL+P, it is a runtime error to apply a strict operation to a persistent value outside the scope of a transaction. A strict operation is one that requires knowledge of a value's contents to complete. Examples are `car` applied to a persistent cons cell, `aref` applied to a persistent array, `get` applied to a persistent symbol, and `eval` if applied to any persistent value. Nonstrict operations do not require value content information. For example `list`, `cons`, and `eq` are nonstrict. Thus persistent values can be passed around inside volatile values and inserted into volatile values while outside a transaction.

3.1 Transaction Related Constructs

`with-transaction` ({*var* | (*var value*)}*) *transaction-form* [*cleanup-form*] [*Macro*]

`with-transaction` introduces a set of bindings. After making the bindings, the `transaction-form` is evaluated; this is the body of the transaction. The transaction succeeds if the body completes execution and if the implicit commit is also successful. Then the result of `with-transaction` is the result of the body. If the form tries to leave the transaction scope via a `go`, `throw`, `abort-transaction`, etc., or if the commit attempt fails, then the transaction aborts. The transaction may be aborted by the runtime system during the course of a transaction if a needed lock is not available (e.g. if a locking transaction protocol were implemented) or if a protocol violation were detected (e.g. when a time-stamping optimistic protocol is in use). The commit attempt can fail because of an I/O error at the server or because the transaction collided with another transaction and failed the validation process (e.g. if the underlying transaction mechanism is commit-time validation).

The commit attempt is made after the completion of the *transaction-form*, and is within the scope of the `with-transaction`. The value of the current binding of any persistent special variable is the value persisted. This may be a binding introduced by the `with-transaction` form. Therefore, although persistent symbols can be used as special variables within the program, only the most recent binding is persisted.

When the transaction aborts, the cleanup form is executed. These forms are outside the scope of the transaction so they cannot reference persistent values. The purpose of the cleanup form is to allow the programmer the opportunity to "roll-back" volatile values that were modified by the transaction body. In the case of an aborted transaction, the result of `with-transaction` is the result of the last cleanup-form.

The variable lisp::*abort-reason* is defined to contain an implementation dependent reason for the transaction abort (e.g. 'deadlock, 'validation-failed, etc.). It can also be set via the `abort-transaction` function (see below).

Support for nested transactions [12] is permitted. An implementation dependent parameter, `*max-transaction-depth*`, specifies the maximum level of nesting. If this level is exceeded then an error will be signalled. The current implementation of UCL+P will only support a single, unnested transaction.

`abort-transaction &optional (code 'user)` *[Function]*

The `abort-transaction` function causes the transaction to be aborted. It also sets the variable `*abort-reason*` to *code*. `abort-transaction` may only be called within the dynamic scope of a transaction or an error will be signalled.

`persistent-p` *arg* *[Function]*

`persistent-p` returns `t` if *arg* is a persistent value and `nil` otherwise. The purpose of this function is to allow the program to determine if a value is persistent and therefore can only be accessed inside a transaction. When used inside a transaction it must be remembered that values that will be made persistent at the end of the transaction are volatile until *after* the commit has succeeded; therefore, `persistent-p` will return `nil` when applied to such values.

`in-transaction-p` *[Function]*

`in-transaction-p` returns the nesting depth (≥ 1) if executed within the dynamic scope of a transaction and `nil` otherwise.

3.2 Examples

Figure 1 shows a simple function to transfer funds between two bank accounts. The function takes three arguments, two account numbers and an amount; all of these values are volatile. The global variable, `accounts`, is a persistent array which contains the balance of each account. The function will transfer *amount* from the source account to the destination account if the source account has enough funds. The function also updates a volatile variable, `*total-handled-today*`, to reflect the amount of funds manipulated during the banking day. When the transfer succeeds, the function returns `t`. If the source account contains insufficient funds, `abort-transaction` is used to exit the transaction. The cleanup form is executed but leaves changes `*total-handled-today*` unchanged due to the `if` statement. If the transaction commit fails, the cleanup form is also executed, but this time it decrements `*total-handled-today*` to reflect the failed operation. The function returns `nil` if the transfer fails for any reason.

```
(defun transfer-funds-1 (src dst amt)
  (with-transaction
      ((left (- (aref accounts src) amt)))
    (cond ((< left 0) (abort-transaction))
          (t (incf *total-handled-today* amt)
             (setf (aref accounts src) left)
             (incf (aref accounts dst) amt)
              t)))
    (progn
      (if (>= left 0)(decf *total-handled-today* amt))
      nil)))
```

Figure 1: Fund transfer example implemented using `abort-transaction`.

```
(defun transfer-funds-2 (src dst amt)
  (catch 'nsf
    (with-transaction
       ((left (- (aref accounts src) amt)))
     (cond ((< left 0)
            (throw 'nsf 's-o-l))
           (t (incf *total-handled-today* amt)
              (setf (aref accounts src) left)
              (incf (aref accounts dst) amt)
              t)))
     (progn
       (if (>= left 0)(decf *total-handled-today* amt))
       nil))))
```

Figure 2: Funds transfer function implemented using `throw`.

Figure 2 shows a different implementation of the fund transfer function. Here, a `throw` is used in place of `abort-transaction`. When the throw is executed, the cleanup forms are executed as in the previous example. However, when the cleanup form is complete, the throw "continues" causing the `'s-o-l` to be returned as the function value.

4 Persistent Packages

The Common Lisp `package` construct was extended to create persistent packages. By providing modularity, standard CL packages assist in the management of symbols and, by extension, their bindings and associated volatile values. Persistent packages extend this support for modularity to persistent values and also underlie our mechanism for conferring persistence onto values. In this section we look at the use, syntax and semantics associated with persistent packages.

4.1 Persistent Packages: Modularity for Persistent Values

By introducing persistence, we have split the value space into two parts: persistent and volatile. A single, large value space is accepted as the norm for volatile values, but for persistent values it is inappropriate. A persistent value space can become enormous, even by Lisp standards, presenting significant management difficulties. These challenges include managing the store (allocation, deallocation, locality, etc.), controlled sharing (at what granularity access can be controlled), and security. Persistent packages make these tasks feasible by modularizing the stored persistent values into manageable subsets much as files partition data on secondary store.

Associating the persistence of values with the package mechanism does not require any significant changes to the programmer's view of Lisp. With few exceptions, the programmer need not be aware of whether a value is persistent or volatile.[1] A symbol is associated with a home package via the usual `intern` process; if the home package is persistent, then so are the bindings of the symbol in force at commit time: its value, its function definition, and its property list. The precise methodology for determining the persistence of values is discussed below (section 5.2).

Persistent packages add a new attribute to packages, the *store-name*. This attribute increases the flexibility of persistent packages by decoupling the actual instance of a package (the store) from its interface (the name of the package and its exported symbols). Since the *store-name* names a file in the host file system, the naming and protection mechanisms of that system are exploited in locating and accessing stores. Even a single user might find it useful to have different persistent packages having the same *package-name*; this is similar to using the `load` function to bring in the appropriate set of forms. For example, a `measures` package could have a symbol `length-unit`. By specifying the *store-name* "mks" the symbol `measures::length-unit` would be bound to "meters", and if the *store-name* "cgs" were used then `measures::length-unit` would be bound to "centimeter". For multiple users, the benefits are even greater, as it is possible to choose private or shared versions of a given package.

During the execution of a Lisp program, persistent packages act very much like volatile packages; a package must be made known to Lisp via `make-package` before any symbols belonging to the package are seen by the reader. Persistent packages do differ, in that some of the package attributes are not persisted.

A persistent package retains the export status of its symbols. If a symbol was internal when the package was saved, then it will still be internal when retrieved. The export status of a symbol can be permanently changed using the Common Lisp functions `export` and `unexport` within the scope of a transaction.

Persistent packages do retain information about the use of other packages, but not the import of specific symbols. The intent is to avoid storing large numbers of off-package symbols that might not actually occur as values within the package. Therefore, any symbols to be imported into a persistent package will have to be explicitly included each time the persistent package is used.

Nicknames for packages present a problem because of the possibility of conflict with the nicknames of other packages. Such a conflict could cause `make-package` to fail. To address this, if `make-package` is provided with a `:nicknames` argument, that argument will override the nicknames stored in the persistent package for the current program execution. A permanent change to the nicknames can be accomplished using `rename-package` within the scope of a transaction.

[1] With the exception that persistent values cannot be accessed outside the scope of a transaction.

4.2 New Syntax and Semantics of Packages

`make-package` *package-name* `&key :nicknames :use :persistent` [*Function*]

`make-package` has been extended with a `:persistent` keyword. The associated argument is the *store-name* for the persistent package (a string). If `:persistent` is not specified, then `make-package` behaves as in standard CL. When `:persistent` is used, one of the following actions will be taken:

1. If a package named *package-name* exists on the store named *store-name*, that persistent package is returned.
2. If no store named *store-name* exists, a new store is created, and a new persistent package is created within that store and returned.
3. If a store named *store-name* exists but contains no package named *package-name*, such a package is created within the store and that package is returned.
4. If any package named *package-name* is already known to the runtime system; a runtime error is signalled.

`make-package` has been changed to return two values. The first is the package created or found on the store. The second value is `t` if a new persistent package was created on the store, and `nil` otherwise.

`in-package` *package-name* `&key :nicknames :use :persistent` [*Function*]

A new keyword `:persistent` was added to the definition for `in-package`. The argument associated with it is the package's *store-name*, either a string or `nil`. When `:persistent` is not used, `in-package` behaves as in standard CL. Note that it may find a previously known persistent package.

When `:persistent` is used with argument `nil`, it behaves as a standard CL `in-package` except that only a volatile package is returned. If a persistent, known package named *package-name* is found, an error is signalled.

When `:persistent` is used with a string argument, a persistent package is always returned. If a volatile package named *package-name* is known, or a known persistent package using *package-name* but having a different *store-name* is found, then an error is signalled. `make-package` is called if there is no known package named *package-name*.

`in-package` now returns multiple values. The first value is the package, and the second value is `t` if a new persistent package was created, and `nil` otherwise (this is always the case for volatile packages).

`package-of` *arg* [*Function*]

The `package-of` function returns the package of its argument. For symbols this is the home package and for persistent values it is the package that contains them. For volatile, nonsymbolic values this function returns nil. The function may be called outside of a transaction.

5 Semantics of Persistence

The semantics of UCL+P are nearly a superset of Common Lisp semantics. Some exceptions were described above: two of the package functions were modified, and transaction semantics dictate that persistent values be accessed only within the scope of a transaction. With these exceptions, all other Common Lisp semantics and syntax hold while values are resident in the executing program.

Common Lisp semantics provide a set of rules to be satisfied during the execution of a program, but naturally have nothing to say about persistent values. Therefore, we have had to devise semantics for persistent values that did not clash with the volatile value semantics and yet made sense for operations on persistent values. In this section we look at the relationship between sharing and the implementation of persistence, at the algorithm for conferring persistence, and at the role of symbols in our persistence schema. Continuing, we examine the mechanism for moving values to and from the store and the approach to references crossing package boundaries. We conclude by exploring a sharing paradox.

5.1 Sharing

Conforming to Lisp semantics requires us to preserve the same sharing semantics between program executions as standard CL does within a single execution of a program. We have maintained Common Lisp sharing semantics for persistent values: if a persistent value is shared by two other persistent values at runtime, that relationship will be preserved when the values are retrieved from the store during later transactions within the same or subsequent program executions. In particular, the property of eq-ness is preserved.

Another approach would be to allow structure sharing to be ignored at commit time when values are persisted. This would lead to some peculiar situations as illustrated in figure 3. Prior to being committed, mutations to a shared value would be visible through all references to the shared object as shown under "Sharing preserved." But mutations in subsequent transactions (or program executions) might be made to copies of the object and would be visible through only a subset of the original references ("Sharing not preserved"). By extending sharing semantics to persistent values, we have avoided this potentially troublesome inconsistency. It should be noted that this preservation of sharing has proved to be one of the most costly aspects of the implementation, due to the fine granularity of Lisp objects. The alternative approach was not rejected lightly.

Preserving sharing semantics required that we provide orthogonal persistence: all data types (with the exception of streams) have been made persistent. Persistence, and therefore sharing, could have been limited to a subset of values (e.g. CLOS objects) available in Lisp, but this would produce a persistent *subset* of Lisp.

5.2 Conferring Persistence

Since Common Lisp does not address persistent values, we had to define the means for creating persistent values. Three approaches were considered. The first, creation-time conferral, is very simple. This requires that all value creating functions (e.g. `cons`, `make-array`, etc.) take an argument to indicate whether the value created is to be persistent. This

```
(in-package 'p :persistent "p.pkg")
(with-transaction ()
  (progn (defvar a (list 1 2 3))
         (defvar b (cdr a))))
```

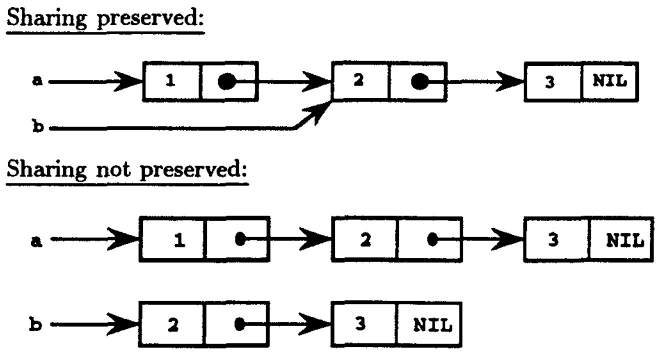


Figure 3: A Sharing Example

approach presents some difficult semantic problems. If an initially volatile value later needs to be made persistent, it would be necessary to make a persistent copy of the value. This would violate the sharing semantics which we intend to maintain.

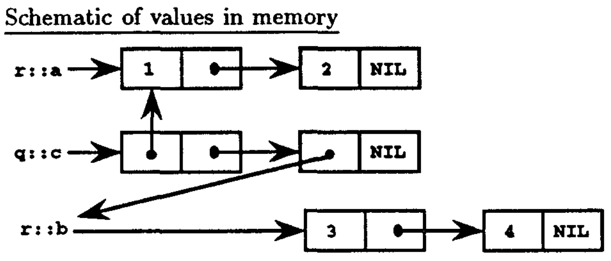
Another scheme for conferring persistence is to explicitly confer it some time after the value was created. A function would be applied to a value to change it from volatile to persistent. The explicitness of this approach does not match up well with the spirit of Lisp.

Both of these first two approaches suffer from the same serious drawback—it is possible to construct composite values (such as graphs of cons cells, arrays, structures, etc.) that contain persistent values which reference volatile values. Such composite values present a problem because references to the volatile portions would be meaningless when such a value was retrieved in a future program execution. To maintain either of these approaches and avoid that drawback would involve changing fundamental operations to have differing semantics based on the persistence attributes of the arguments provided. This violates our premise that persistent and volatile values should possess identical semantics within a given program execution.

Our approach, persistence by reachability, conforms to both the law and spirit of Lisp. Using reachability, a non-symbolic, volatile value[2] becomes persistent at commit time if a reference path passes from a persistent value to it; reference chains do not pass *through* symbols via their bindings, though they will always originate at a persistent symbol. The roots for this conference are the symbols within the persistent package(s). The concept is analogous to the determination of liveness for garbage collection. Figure 4 shows an example of this approach. The body of the transaction contains three statements which intern the variable "c" into persistent package "q" and set up in memory the situation illustrated by the memory schematic. At commit-time, `q::c` serves as the roots for the conferral algorithm, being the

[2]Packages are either volatile or persistent based on how they are created. In the unlikely event that a package becomes part of a persistent value (e.g. (defvar per::w (find-package 'lisp))) the reference will be stored as a reference to a package named "LISP".

```
(make-package 'q :persistent "q.pkg")
(in-package 'r)
(with-transaction ()
  (progn (defvar a (list 1 2 ))
         (defvar b (list 3 4))
         (defvar q::c (list a 'b))))
```



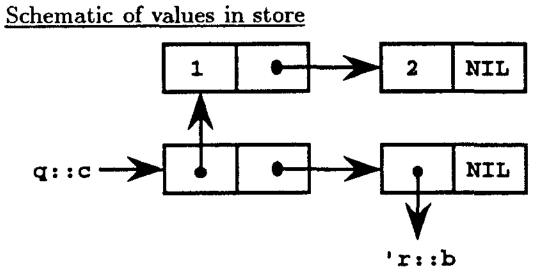


Figure 4: Conferring Persistence

only persistent symbol. Reachable from this root are the cons cells making up (list a 'b), the *value* of `r::a`, `(list 1 2)`. The bindings of `r::b` are not made persistent because `q::c` references only the symbol, `'r::b`, not its bindings.

Because our approach does not affect the eq-ness of the values, all sharing semantics are preserved. Since the process is automatic, functions need not be aware of whether their arguments are persistent or volatile as there is no risk that persistent values will be left with dangling references to volatile values.

The one drawback of this approach is that persistent values may inadvertently be incorporated into previously volatile data structures. If this occurs an error will be signalled if the persistent component of the data structure is accessed outside the scope of a transaction. While this is certainly not a pleasant occurrence it is a type of fault that is readily detectable, whereas dangling references placed into the store produce a persistent minefield!

Another disadvantage can occur if persistent data structures contain temporary data such as intermediate results in a large computation. Since persistence is conferred by simple reachability, the system cannot distinguish the intermediate results from the final results and will save both. The programmer needs to either design the data structures such that they do not contain unneeded data or that the *trash* is dumped before committal (e.g. by putting `nil` into slots containing temporary data).

5.3 Symbols

The symbol is a very important and unique feature in CL. Three of the symbol properties are important to understanding the semantics of persistence for symbols. First, symbols can serve as unique values, in and of themselves, which make them very useful for symbolic computations (e.g. `(if (eq account-state 'frozen) (reject-login))`). Second, symbols can be bound or associated with several different values which can extracted when needed allowing symbols to serve as named variables, named functions, and more miscellaneous roles. Lastly, unlike other values, interned symbols are not subject to garbage collection; once created they exist unless explicitly uninterned. These properties guided us in our definition of a persistence semantics for symbols.

When symbols are created, they are interned into their home package. Whether the symbol is persistent or volatile depends on whether that home package is persistent or volatile. Unlike other values, an interned symbol[3] never changes from volatile to persistent when referenced by a persistent value. The reasons for this are several. When a nonsymbolic value is made persistent, it becomes part of the persistent package conferring persistence onto it. This is necessary to prevent the store from containing values with references to nonexistent, volatile values. To apply an analogous approach to symbols would require either that a referenced volatile symbol be made a part of the persistent package, or the home package of the referenced volatile symbol be made persistent as well. Moving the volatile symbol into the persistent package would be a violation of package semantics. Conferring persistence onto the containing package would effectively result in all packages eventually becoming persistent requiring all data accesses to follow transaction semantics.

Fortunately, neither of these two dire solutions are necessary. Although symbols can be *evaluated* to values (or to functions), they also stand alone as values. Therefore when a persistent value contains a symbol, a dangling reference is not created. When the contained symbol is only used as the subject of a symbolic computation, the containing persistent value is complete. In defining the semantics of persistence for symbols when they are subject to evaluation, they are intentionally treated as interfaces to the values and functions of other packages and, as such, the program must provide the appropriate bindings at runtime. This is consistent with the Lisp model for symbol use: they provide a level of naming that allows for dynamic rebinding. Therefore volatile symbols referenced by a persistent value do not become persistent. The containing persistent value retains the symbol's name for reconstruction whenever the persistent value is retrieved from the store.

This approach cuts two ways. It places a burden on the user of a persistent package: a meaningful set of bindings must be provided for any symbols external to that package which appear as values within it. On the other hand, it provides the user with the flexibility of choosing appropriate bindings for those symbols at each use the persistent package.

A pragmatic reason for selecting this approach is that persisting the bindings of symbols external to the persistent package could transitively include extensive portions of, if

[3]Uninterned symbols are a special case. Whether originally persistent or volatile, uninterned persistent symbols follow the same persistence conferral semantics as other values.

not the entire, execution state. This would constitute a gross violation of our goal of modularity.

An issue introduced by persistence is that of circular references. In standard CL there is no difficulty in creating interpackage references, wherein the *values* bound to symbols in the two packages each contain the other symbol. If these two packages both persist, a problem could arise in attempting to instantiate a value in one package that consists of a symbol in a package that has not yet been made known. This is a case where lazy loading (see next section) has not only performance implications but provides a necessary support for the semantics of the persistence.

5.4 Fetch and Store

A persistent language should provide more power than *ad hoc* approaches based on the I/O facilities provided by the language. It should ease the task of the programmer in writing programs that manipulate persistent values. An approach that requires the program to explicitly retrieve values from the store is not much more powerful than the read/write operations persistent languages seek to supplant. Similarly, requiring the programmer to mark those values which are modified so that they will be stored at commit time imposes a large burden on the programmer. If the programmer neglects to write even a single changed value back to the store, it can leave the persistent store in an inconsistent state.

UCL+P provides transparent fetch and store of persistent values. The runtime system assures that values needed from the store are fetched prior to their use and that mutated/created values are automatically written back to the store when a transaction commits. Further, the language semantics specify lazy loading of persistent values. At first glance this would seem to be an implementation issue and irrelevant to the language semantics; however, there is an indirect semantic effect. Since persistent packages can become quite large and because a program may simultaneously use several persistent packages, it is possible that a program might fail due to excessive memory requirements if an eager loading approach to fetch were defined. The lazy approach defers unnecessary loading, and permits programs to execute successfully so long as the values they actually use will fit in memory.

Under lazy loading, when referencing a persistent package by `make-package` or `in-package`, only the package symbol table is initially fetched from the store. As nonresident values of the package are accessed, the runtime system retrieves them from the store. Only accessed values are actually loaded into the program.[4] This lazy loading approach allows manageable subsets of very large persistent packages to be manipulated.

Figure 5 illustrates the lazy loading feature. When persistent package "x" is opened, all of its symbols are loaded; one of these symbols, "a" was bound to `(list 1 2 3)` in a previous program execution. At the start of the transaction, only the symbol itself is memory resident, as depicted in the first schematic. As each of the three `format` statements cause the list to be walked, more of the list is made memory resident.[5]

[4] Other values located on the same storage block may also be made resident. This is a detail of the store manager implementation.

[5] Note, however, that an efficient store manager would attempt to place all three cons cells on the same store page to exploit locality, and therefore all three cells might well be loaded simultaneously. If the values were very large vectors they could not reside on the same page and the data structure would certainly be fetched incrementally.

```
(in-package 'x :persistent "x.tmp")    ; A
(with-transaction ()
  (progn (format t "~a~%" (car a))     ; B
         (format t "~a~%" (cadr a))    ; C
         (format t "~a~%" (caddr a)))) ; D
```

Values fetched at each step:

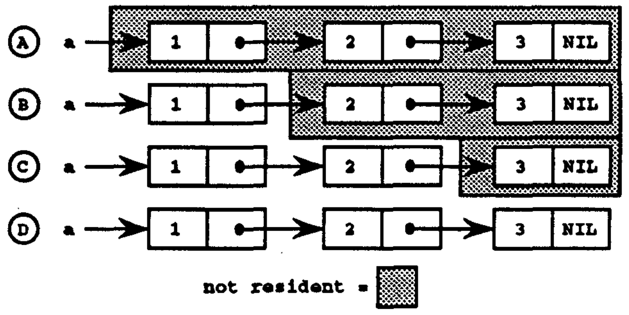


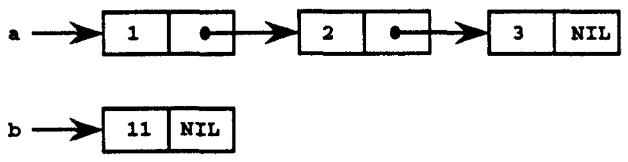
Figure 5: A Fetching Example

All persistent value accesses are automatically detected and recorded, to support store conflict assessment at transaction commit time. In addition, detection of write accesses enables automatic determination of which values must be saved to the store at commit time. Newly created values become part of the mutated set either by virtue of being added to an existing persistent object (value mutation) or by being assigned as the value of a persistent symbol binding (symbol mutation). An example is shown in figure 6. Persistent package "x" contains two symbols, "a", and "b", bound to `(list 1 2 3)` and (list 11) respectively; the first schematic depicts these values. The transaction body causes a destructive mutation of the list bound to "a", and conses a value onto the front of the value of "b", then binding "b" to the new value. The second schematic shows the new values; the shading marks the mutated values which are written back to the store during the commit process. The unshaded values are already known to the store and are not rewritten, thereby saving translation effort by the runtime system, IPC bandwidth, and processing by the store manager.

5.5 Interpackage References

The combination of sharing semantics, and simultaneous use of multiple persistent packages leads to the possibility of interpackage references. Although in most cases, each value is properly contained within a single package, UCL+P does support cross-package shared structures (e.g., when values from two distinct packages share the tail of a list). The shared portion of the structure can only reside in one store. In the package not containing the shared portion of the structure, an *interpackage reference* to the shared portion is stored. The reference denotes the persistent package (via its *store-name*) and a reference to the structure within that store. While interpackage sharing of structured values is supported, using symbols for interpackage references is a better approach since it makes the dependence on another package both explicit and potentially more flexible.

Values already on store:



```
(in-package 'x :persistent "x.tmp")
(with-transaction ()
  (progn (nconc a (list 4))
         (setf b (cons 10 b))))
```

Values sent to store are shaded.

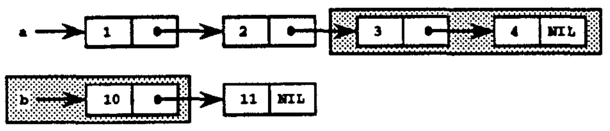


Figure 6: A Storing Example

Interpackage symbol references are trivial. The symbol itself in this case is the value; only the full symbol name is saved. When a persistent value containing a symbol reference is loaded, the stored symbol name is used to establish a reference to the symbol currently bound to that name in the current environment. A symbol reference in a persistent value can resolve to a symbol in a persistent package in one program execution and to a volatile symbol in another. As with all symbol references, the package containing the referenced symbol must be known to Lisp.

5.6 A Sharing Paradox

Sharing semantics are preserved across both persistent and volatile values during the execution of a program, and between persistent values for their entire lifetime. However, when a subsequent program execution retrieves persistent values from the store, the sharing relationship may not be as first expected. Consider the example in figures 7 & 8. During the first program execution, variable a is created and bound to the value `(1 2)`. Variable `per::b` is created and bound to the value of a. Not surprisingly, the values of both variables are eq. On a subsequent program execution (figure 8), a is again bound to `(1 2)`, but the value is no longer eq with the value of `per::b` as returned from the store. Although this might be surprising at first, the lack of eq-ness follows from the fact that in this execution a new constant `(1 2)` was created for the value of a and, as such, can not be expected to have an eq relationship with the constant created in the previous program execution. The situation is analogous to one in Common Lisp:

Given: `(defvar a '(x y))`
`(defvar b '(x y))`

we cannot conclude:

`(eq a b)` will return `t`.

```
(make-package 'per :persistent "per.pkg")
(in-package 'user)
(defvar a '(1 2))
(with-transaction ()
  (defvar per::b a))
```

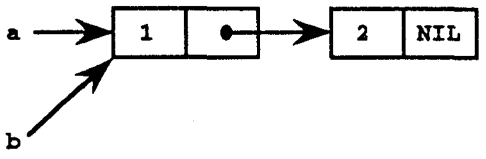


Figure 7: Sharing paradox: first program execution.

```
(make-package 'per :persistent "per.pkg")
(in-package 'user)
(defvar a '(1 2))
(with-transaction ()
  (eq per::b a)) ; ==> nil
```

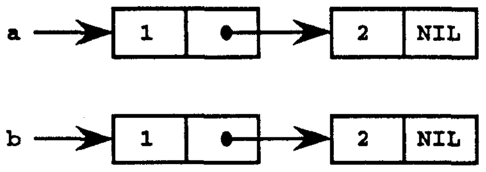


Figure 8: Sharing paradox: second program execution

6 Persistent Semantics of Other Data Types

In the previous section we covered the major semantic issues of persistence and we have detailed the persistence semantics of the types, packages and symbols. However, Lisp is a language rich in data types and the discussion of persistent semantics would be incomplete without examining the rest of the data types. For some of the data types (e.g. numbers) the semantics of persistence is quite straight forward, while for others (e.g. functions) it is fairly complex.

6.1 Numbers, Characters

The Common Lisp definition of eq-ness for numbers and characters permits the implementation to use copying instead of sharing [16]. Therefore no Lisp program can safely rely on two numbers being eq to each other. The information contained in a numeric or character value is preserved in UCL+P.

6.2 Streams

Values of type stream are not allowed to persist. This is because the ultimate stream values (e.g. files) are in the domain of the host operating system and follow semantics defined by it.

6.3 Lists, Arrays, Hash Tables, Readtables, Pathnames, and Random States

Lists, arrays, hash tables, readtables, random states, and pathnames are all fairly simple aggregate data types. Because persistence preserves the semantics of sharing, these data types all perform as would be expected; eq-ness is preserved.

6.4 Functions

Functions, like any other Lisp value, can become persistent. One questions arises: how do functions participate in the persistence conferral scheme?

Functions confer persistence on values and other functions via the reachability algorithm described earlier. It is important to note that the named functions called by a function are referenced via symbols. As described earlier in Section 5.3, the symbol is persisted, not its function definition. The same applies to value references to global symbols.

The programmer is required to guarantee that any named helper functions used by a persistent function, but which are external to the persistent package, be defined before use of the persistent function. They may either be volatile and therefore have been made present by previous `load`'s or they may reside in other persistent packages which must be made known by `make-package`'s. In the latter case, the actual definition of the helper function may actually not be present until a call on it occurs.

Anonymous functions referenced by persistent values are automatically persisted in accordance with the reachability algorithm. Anonymous references to functions can be created in a variety of ways. They can be created by using `lambda`, `labels`, and `flet`, or by calling `symbol-function` which returns an anonymous reference to a named function. Automatic persistence for anonymous functions is not only consistent with the conference policy, but necessary, since there is no way for a user to provide the needed function in future executions.

Persistence of closures requires that their environment be stored along with the code. Logically, a closure can be thought of as a pair of function and values (environment). Each part of the pair is handled according to the rules for functions and values.

Figure 9 shows the difference between saving a function's name and saving its code. The transaction defines two symbols, "f" and "k", within persistent package "p". The body of function "f" calls the function `z::g`. The symbol function slot of "k" is bound to the symbol function of "g". The values persisted are illustrated in the schematic. Since "f" calls `z::g` by name, the persisted code for "f" will contain a reference to the *symbol* `z::g`; no direct reference to the function for `z::g` is persisted. The situation is a little different for "k". Since "k" is bound to the same function that `z::g` is bound to, the function value of "k" will contain a reference to the code for function "g". When a symbol is evaluated to obtain any of its bindings, the relationship between resulting value and the symbol is lost and cannot, in general, be reconstructed. Therefore, the value itself must be persisted.

6.5 Structures and Conditions

Structures are a fairly complex data type. We can split structure operations into two categories: type and instance.

```
(in-package z) ; not persistent
(defun g (x) x)
(in-package 'p :persistent "p.tmp")
(with-transaction ()
  (progn (defun f(x) (z::g x))
         (defvar k)
         (setf (symbol-function 'k) #'z::h)))
```

Schematic of values on the store after commit.

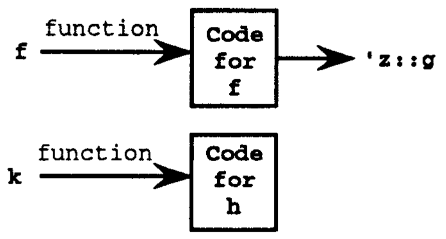


Figure 9: Result of persisting functions.

The `defstruct` operation creates a named data type together with a set of functions: slot accessors, constructor, predicate, etc. Each of the functions created is bound to a symbol. Once the data type is defined, instances can be created and manipulated. Given a structure instance, the runtime system can determine its data type and size directly. With these two pieces of information the store manager is able to save and retrieve a structure value. However, without the information provided in the `defstruct` statement, a program can only treat the structure as an opaque value.

To fully use a structure requires all of the auxiliary functions. Because these are named functions they derive their persistence status from the naming symbol. It falls to the programmer to ensure that the needed functions are available, either by defining them within a persistent package or by providing them as volatile functions.

When a structure instance is retrieved from the store it is possible that the current data type definition for it is different than when it was created. This is analogous to performing a sequence of `defstruct`, `make-struct`, `defstruct`. The Lisp standard states that the results of redefining a structure are undefined [16] in the volatile world, so the example described above for persistent values is compatible with similar volatile value semantics.

The data layout and behavior of `conditions` are comparable to structures and are handled analogously.

6.6 Persistence Semantics of Classes, Methods, and Generic Functions

These data types were introduced into Common Lisp along with CLOS. CLOS provides a very powerful object system, including multiple inheritance, the meta-object protocol, and a simple form of type evolution. Persisting the data components of individual objects is straightforward, as a consequence of our orthogonal approach to persistence. The issues related to persisting methods are largely analogous to those for functions and structures. But subtle issues arise with the interaction of the complex semantics of CLOS objects and the state it requires within the runtime system.

There are also issues relating to the interaction of persistence with type evolution and with time-specific methods such as initializers. We have not yet concluded the process of crafting a persistence semantics for CLOS.

7 Conclusions

We believe we have that in UCL+P we have created a Persistent Common Lisp which fully supports persistence while maintaining the look-and-feel of Common Lisp. The transaction mechanism allows the programmer to easily construct programs that will leave the store in an consistent state even when the system fails or when the store is concurrently accessed by other programs. The ease with which persistent Lisp programs can be developed is due in large part to the preservation of the Lisp style of programming. Sharing semantics are preserved between *all* values during any program execution and *at all times* between persistent values. Persistence is automatically conferred onto nonsymbol values when they are needed as part of a persistent data structure, while symbols are persistent if they reside in a persistent package. Needed persistent values are automatically retrieved from the store on demand; new or mutated values are implicitly detected and automatically written to the store.

Besides serving as the ultimate root for our persistence conferral algorithm, persistent packages are important from the store perspective. By providing fairly self-contained containers of persistent values, persistent packages modularize the store making the persistent store management a much more tractable endeavor.

Our successful definition of a persistent Lisp is due in large part on our willingness undertake all necessary measures to implement the language definition. We went into the guts of the UCL compiler and runtime system to implement our definition. This freed us from any undue concerns about being able to realize an efficient implementation of persistent Lisp and would not have been possible if we had to implement persistent Lisp as a nonintegral layer on top of Common Lisp. A Lisp aware store was also designed to efficiently support the needs of the language. The low-level approach also allows for further optimizations in both the compiler and store manager.

From a language design and implementation viewpoint we believe that UCL+P is a success, but until several independent Lisp programmers have created some significant application programs in UCL+P, we will not be able to fully evaluate the success of the project. As with all languages, no matter how well designed, only time will tell if this effort is more than an academic exercise.

References

[1] R. Agrawal and Gehani N. H. ODE (Object Database and Environment): The language and data model. In *Proc. Int'l. Conf. on Management of Data*, pages 36–45, Portland, Oregon, May-June 1989. ACM-SIGMOD.

[2] Gilles Barbedette. LispO_2: A persistent object-oriented LISP. In F. Bancilhon, C. Delobel, and P. Kanellakkis, editors, *Building an Object-Oriented Database System: The Story of O_2*, chapter 10, pages 215–233. Morgan Kaufmann, 1992. Also in Proceeding of the 2nd EDBT.

[3] P. Broadbery and Burdorf C. Applications of Telos. *Lisp and Symbolic Computation*, 6(1/2):139–158, August 1993.

[4] W. P. Cockshott. *PS-ALGOL Implementations: Applications in Persistent Object-oriented Programming.* Ellis Horwood, 1990.

[5] Korth H. F. and A. Silberschatz. *Database System Concepts, 2e.* McGraw-Hill, 1991.

[6] S. Ford, J. Joseph, Langworthy D., D. Lively, G. Pathak, E. Perez, R. Peterson, D. Sparacin, S. Thatte, Wells D., and S. Agarwala. Zeitgeist: Database support for object-oriented programming. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems.* Springer-Verlag, 1988.

[7] R. P. Gabriel. *Perfomance and Evaluation of Lisp Systems.* MIT Press, 1985.

[8] A. L. Hosking, J. E. B. Moss, and C. Bliss. Design of an object faulting persistent Smalltalk. Technical report, Univerity of Massachusetts, 1990. UM-CS-1990-045.

[9] J. H. Jacobs, M. R. Swanson, and R. R. Kessler. Persistence is hard, then you die! or Compiler and runtime support for a persistent common lisp. Technical report, Center for Software Science, University of Utah, 1994. UUCS-94-004.

[10] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, Oct 1991.

[11] Arthur H. Lee. *The Persistent Object System MetaStore: Persistence via Metaprogramming.* PhD thesis, University of Utah, Aug 1992.

[12] J. Eliot B. Moss. Nested transactions: An introduction. In Bharat K. Bhargava, editor, *Concurrency Control and Reliability in Distributed Systems*, chapter 14, pages 395–425. Van Nostrand Reinhold, 1987.

[13] S. M. Nettles and Wing J. M. Persistence + undoability = transactions. In *Proceedings of the Hawaii International Conference on Systems Science 25*, 1992. See also tech-report CMU-CS-91-173.

[14] A. Paepcke. PCLOS: A flexible implementation of CLOS persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming.* Springer-Verlag, 1988.

[15] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. Technical report, University of Wisconsin, 1989. Tech Report 824.

[16] Guy L. Steele, Jr. *Common Lisp: The Language, Second Edition.* Digital Press, 91.