A Framework for Maintaining the Coherence of a Running Lisp

Drew McDermott
Yale Computer Science Department
P.O. Box 208285
New Haven, CT 06520-8285
drew.mcdermott@yale.edu

Keywords: Inference, algorithms, consistency.

Abstract

During Lisp software development, it is normal to revise and reload programs and data structures continually. The result is that the state of the Lisp process can become "incoherent," with updates to "supporting chunks" coming *after* updates to they chunks they support. The word *chunk* is used here to mean any entity, content, or entity association, or anything else modelable as up to date or out of date. To maintain coherence requires explicit management of an acyclic network of chunks, which can depend on conjunctions and disjunctions of other chunks; further, the updating of a chunk can require additional chunks. In spite of these complexities, the system presented in this paper is guaranteed to keep the chunk network up to date if each chunk's "deriver" is correct, the *deriver* being the code that brings that chunk up to date.

Lisp novices are often surprised to find out that Lisp is a "shell" as well as a compiler. Unlike C++ or Java, one starts Lisp, loads functions and data into it, and plays around with them. The advantages of this architecture are well known: it's easy to modify the system incrementally — and experimentally; it's easy to combine two or more programs; the debugger is easily integrated with the rest of the run-time system; and so forth.

Because the typical Lisp session lasts a long time, the programmer must worry about keeping it in a "good" state. Tools for maintaining "goodness" incude features such as `unwind-protect`, which ensure that even when a program bombs its sensitive side effects can be undone. Although the same functionality can be found in other languages (e.g., Java's `finally` clauses), only in Lisp is it a routine tool used by the programmer to ensure that the current core image can continue to execute in what I will call a *coherent* state. Another example is the distinction between `defvar` and `defparameter`, which would be meaningless in most languages, but in Lisp is crucial to making sure that one can reload a file while undoing "just the right set" of global-variable assignments since the file was last loaded.[1]

However, the built-in facilities of Lisp do not address the coherence problem in any systematic way. For example, although it is easy to reload a file after making some bug fixes, it often happens that the reloaded file initialized some table, and entries were made in it by files loaded later. The other files must then be reloaded, unless that causes further glitches. Sometimes one can use `defvar` to avoid reinitializing the table, but sometimes that's simply inadequate; some of the later entries are to be retained and some discarded, and there's no obvious way to sort them out. Often one must resort to restarting Lisp and reloading the program's files in the original order.

There is nothing really wrong with restarting. It often requires you to take special measures to get back to the point you were at before the restart. Every Lisp programmer has had to build "restart scripts," sequences of expressions whose evaluation will get the Lisp back to the middle of a debugging sequence. Aside from this nuisance, it seems as if there ought to be a way to formalize the dependencies among the parts of a Lisp "session" in such a way that revising one part causes the other parts to revise themselves to restore coherence. Rather than maintaining a collection of ad-hoc restart scripts, one could instead assert

[1] Lisp is not entirely alone in having coherence issues. Prolog makes an analogous distinction in its distinguishing between loading and reloading a file. And, of course, Prolog also has the analogue of a read-eval-print loop.

the relationships among parts explicitly and permanently. The explicit statement allows anyone reading the code to understand how the parts relate. The fact that the assertions are permanent means that as further system development occurs, the smooth functioning of pieces developed earlier can be taken for granted.

One symptom of the absence of such a formal theory of *session coherence* is how hard it is to get `defsystem` right.[2] `defsystem` is, of course, the Lisp world's analogue of `make`. There are several versions of it. Some are complex, some simple. Some are "procedural" and some "declarative." The former are more like `make` in that they mainly organize sequences of actions in the space of compiling and loading files. "Declarative" systems attempt to define a system as a collection of files on which different operations can be defined. The popular ASDF [5] seems to have caught on quickly because it is cleanly written and supplies a nice distributed package-management facility. None of these systems address the issues dealt with here, including how to keep a system coherent as different versions of files are loaded.

The present paper may be related to work on persistent objects [1, 2], in that it connects entities in process memory to entities in long-term storage. It is possible that such a facility might be useful as a foundation for the system I describe, but, as we will see, the real problem is getting the logic right. The details of how objects and their interdependencies are implemented are not that crucial.

1 Chunks

We introduce the term *chunk* to mean a piece of information in a particular state, form, or location. Examples of chunks are

- A file
- A Lisp file as loaded into memory in an executable form
- In a table of S-expression handlers, the subset corresponding to executable Lisp expressions whose `car`s are one of the built-in Lisp special operators
- An association between two directories S and C, such that object files compiled from source files in S belong in C.

The components of the Chunk data structure will be unfolded as this paper progresses. However, one thing that will in general not appear as part of a chunk is the entity that it keeps track of. For one thing, there may be no such entity. The chunk for "File `foo` loaded into memory" does not correspond to any information not found in `foo`. If a chunk depends on no other chunk it is said to be *given*; otherwise, it is *derived*. The information associated with a given chunk is set from outside the system, and so is likely to be describable by a noun phrase, such as "Contents of file `F`." If no noun phrase is applicable, as is usually the case with derived chunks, I will use a declarative phrase, such as "File `F` is loaded." Either way, we say the chunk *manages* the noun phrase or the statement: "Chunk `C` manages 'File `F` is loaded'." If derived chunk C manages P, then C is up to date when and only when P is true. A given chunk, by contrast, is up to date when and only when the system *knows* the date when the information it manages last changed. In general, the goal of the coherence system is to make sure that all[3] chunks are up to date

A chunk can obviously be almost any piece or state of information, but the intent is that it be "largish." If a spreadsheet cell is supposed to hold the total of a column of cells, that could be analyzed as a chunk

[2] See [4] for a discussion of some of the issues involved in the design of system-description macros.

[3] I'll qualify this quantifier shortly.

(managin "The total of those numbers is stored in that cell"), but the mechanisms I propose may not be cost-effective for something so small, especially if the chunks change frequently.[4]

I introduce the generic function `derive` that brings a chunk up to date: For a derived chunk, `(derive` c`)` computes something, moves something around, translates something, or does some other transformation of data. For a "given" chunk, `derive` just verifies the date when the content the chunk manages last changed. All the chunk-management system knows is what is revealed by the return value of `derive`. If it returns `nil`, it has determined that the chunk is already up to date. If it returns a number > 0, that means that it or someone else changed something, and that the number is the exact time when the change occurred. Time can be measured in whatever scheme is convenient (so long as it's expressible by a number, and so long as all the chunks connected to each other use the same scheme).

On some occasions it is useful to determine the date of a chunk without running its deriver (i.e., the method supplied for `derive` applied to chunks of this class). The generic function `(derive-date` c`)` finds the date at which c was last derived, if possible. When the date is not available, the deriver may return `nil`, meaning that one should assume the previously stored date to be accurate, or the constant `+no-info-date+` (an integer < 0, and hence not a legal date), meaning that there is no way to know the date without calling `derive`. Which of these is appropriate depends on exactly what the chunk manages.

An obvious, and classic, example of a *dependency* among chunks is the relation between an object file and its source file. When its source file changes, the object file is out of date. We identify two chunks here (to start with): one managing the source file and one managing "The object file is the result of compiling the source file." If the source file changes, then the compiled file must be rederived; that is, `derive` must be applied to it, and the method for `derive` must call the compiler to recompile it. We use the term *basis* for the set of chunks that a given chunk depends on in this way: If chunk F is in the basis of chunk G, then if F changes G must be recomputed — or placed anew, or transformed, or whatever operation corresponds to G's being "up to date." G is said to be a *derivee* of F.

There is one escape clause, however. A chunk can exist but be dormant, in the sense that the chunk system is not required to track it. Application programs tell the chunk system when to flip the chunk from dormant to *managed,* the term I'll use for a chunk that the chunk system must keep up to date. I'll return to this issue in section 2.

Dependencies are of three sorts:

1. *Conjunctive:* This is what is captured by the chunk's basis. If the basis of C is $\{B_1, \ldots, B_n\}$, then C must be changed whenever some of the B_i have changed, but only after all have been brought up to date.

2. *Disjunctive:* A special class of Chunks are the *or-chunks*. In addition to a basis, such a chunk has a non-empty set of *disjuncts*. The or-chunk is up to date if one of its disjuncts is (in addition to its basis).

3. *Transient:* Sometimes a chunk C is not dependent on chunk R, but cannot be updated unless R is up to date. The set of such R's are said to be the *update basis* of C.

The chunk network of figure 1 provides an example. Chunk $C2$ represents the compiled version of file `file2.lisp`, itself represented by chunk $F2$. In addition, $F2$ uses macros defined in file `file1.lisp`, represented by chunk $F1$. That means $C2$ depends on the chunk $M1$ =`(:macros file1.lisp)`, which represents the macros in $F1$. Hence, `file2.lisp` needs to be recompiled if either `file2.lisp` or $M1$ changes. In chunk jargon, $C2$, if it is managed, seems to have as its basis $\{F2, M1\}$. ($M1$, in turn, has $\{F1\}$ as its basis.)

[4] Tilton's [6] `CELLS` system provides spreadsheet-like functionality in Lisp. The issues that arise in that application are, as we will see, not the same as the ones I am talking about.

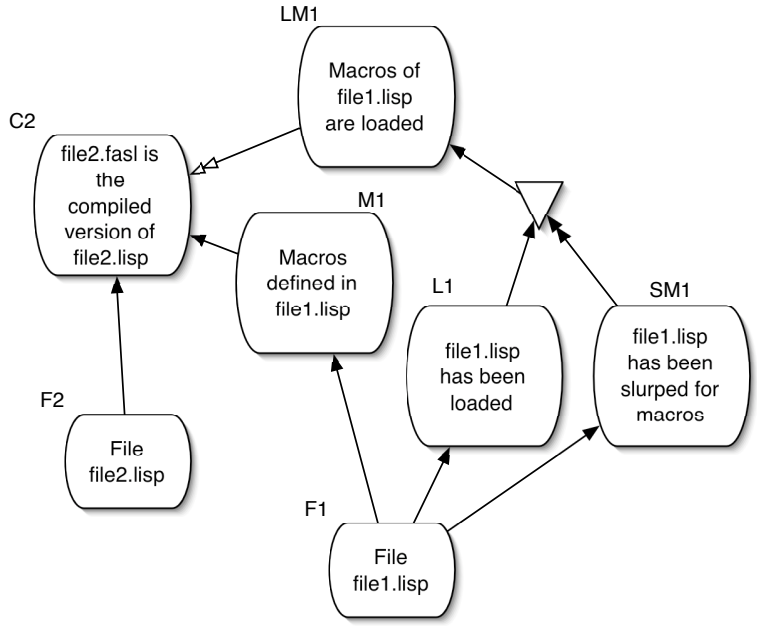

Fig 1: Dependencies among file chunks

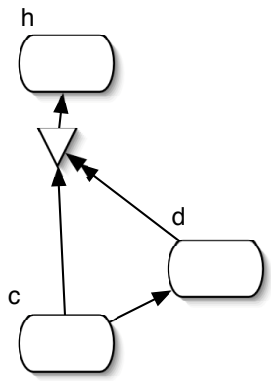

Fig 2: Unstable dependency cycle

But that's not quite adequate. If `file2.lisp` actually needs to be recompiled, it is not enough that $M1$ be up to date; it is also required that the macros in `file1.lisp` be loaded into the running Lisp. We introduce a new chunk $LM1$ =`(:loaded (:macros file1.lisp))`. $LM1$ can be brought up to date by going through the file and evaluating all the macro definitions it contains. I'll call this *slurping* the file. It's unusual to make use of a slurper in the Lisp world, but bear with me. $LM1$ must be up to date in order to bring $C2$ up to date, that is, in order to compile `file2`. So the set $\{LM1\}$ is the *update basis* of $C2$; this is a transient dependency (indicated by an arrow with a double white head).

We go on to observe that, if `file1.lisp` or its compiled version has been loaded, it is unnecessary to slurp it. So $LM1$ must be an or-chunk, with two chunks as disjuncts: $L1$ =`(:loaded file1.lisp)` and $SM1$ =`(:slurped (:macros file1.lisp))`. The second is marked as the *default disjunct* of $LM1$ (indicated by the double-headed black arrow pointing to the triangle indicating disjunction). To ensure that a managed or-chunk always has a *current selection* (a managed base), we require that every or-chunk have a default disjunct, the one that gets managed if none of the others are.

The default method for `derive` applied to an `Or-chunk` is instructive:

```
(defmethod derive ((orch Or-chunk))
   (let ((date nil))
      (dolist (d (Or-chunk-disjuncts orch)
                (or date
                    (error
                        "No disjunct of or-chunk ~s is managed and up to date"
                        orch)))
         (cond ((and (Chunk-managed d)
                     (chunk-up-to-date d))
                (cond ((or (not date)
                           (and (< (Chunk-date d) date)
                                (>= (Chunk-date d) 0)))
                       (setq date (Chunk-date d)))))))))
```

Subclasses of `Or-chunk` may need to do more, but a "bare OR" simply represents that one of its disjuncts is up to date. The method just searches through the disjuncts checking the dates of the managed, up-to-date disjuncts, and returns the date of the one brought up to date earliest. It turns out that we don't need another slot in the `Chunk` class to keep track of its current selection; we just store a singleton list with its current selection as the chunk's update basis. The semantics are exactly as required: that the selection be up to date at the point where the or-chunk is derived.

It would nice if we could insist that every `derive` method be *purely local*, in the sense that it does absolutely nothing except bring the chunk up to date, and in particular does not change the chunk network or the state of any chunk besides itself. Unfortunately, in realistic systems some chunks' purpose is to mess around with other chunks. For example, if a file F specifies in its header what other files are required to be loaded before it is loaded, then the chunk managing "F's header is loaded" will alter the basis of the chunk managing "file F is loaded."

In the rest of the paper, I will use the following terminology. An *immediate supporter* of a chunk C is an element of its basis, or its default disjunct if C is an or-chunk. The *supporters* of C are those chunks related by the transitive closure of the "immediate supporter" relation.

The *height* of a chunk is then defined in the obvious way. If the chunk is not an or-chunk and has an empty basis, then it is called a *leaf chunk*, and has a height of 0. Otherwise, its height is 1 + the height of its highest immediate supporter.

2 Keeping Track of Managed Chunks

Each chunk contains two binary flags, `manage-request`, which keeps track of whether the user has requested that the chunk be managed; and `managed`, which is true if `manage-request` is true, or if it is necessary to manage this chunk in order to manage one of its derivees. I will use the letter R to abbreviate the `manage-request` markers on chunks. A marking R is a function that assigns $R(c) = \texttt{t}$ to a chunk c if and only if the user has requested that c be managed. A marking M is a similar function: $M(c) = \texttt{t}$ iff c is managed.

If $M(c) = \texttt{t}$, a *local cause* of $M(c)$ with respect to a given M and R is one of three things:

1. c itself, in the case where $R(c) = \texttt{t}$;
2. a chunk d such that c is an element of the basis of d and $M(d) = \texttt{t}$;
3. or an or-chunk h such that c is the default disjunct of h, $M(h) = \texttt{t}$, and for every other disjunct c' of h, $M(c') = \texttt{nil}$.

A *supporting path* for $M(c_n)$ with respect to M and R is a sequence of chunks $c_0, c_1, \ldots, c_n$ such that $M(c_i) = \texttt{t}$ for all $i \in [0, n]$, $R(c_0) = \texttt{t}$, and for all $i \in [0, n-1]$, c_i is a local cause of $M(c_{i+1})$. (n may $= 0$.) A supporting path $c_0, \ldots, c_n$ is always in the "down" direction: c_i has height greater than c_{i+1}, and a change in the management status of c_i can cause a change in the management status of c_{i+1}, but never vice versa.

An M is a *closure* of R if

$$\{c | M(c) = \texttt{t}\} = \{c |\ \textit{there is a supporting path for } M(c) \textit{ with respect to } M \textit{ and } R\}$$

There is in general more than one possible closure. A key job of the chunk-management system is to find one of them. It is carried out by two mutually recursive programs, `chunk-manage` and `chunk-unmanage`, which are called to bring the management flags back to closure after the user calls `chunk-request-mgt` or `chunk-terminate-mgt` to change the `manage-request` flag of some chunk.

`(chunk-manage` c`)` is called whenever a local cause for c to be managed is detected. `(chunk-unmanage` c`)` is called whenever the last local cause for c to be managed is removed. `chunk-manage`'s essential task is to make sure that the basis of a managed chunk is managed, which is a simple recursion. `chunk-unmanage` checks to see if ceasing to manage chunk c removes the last local cause for some of the chunks in its basis, and if so unmanages them as well. These simple recursions are made more complex by the existence of or-chunks. Suppose a chunk c becomes managed, at which point it is the only managed non-default disjunct of or-chunk h. Call the default disjunct d. If h itself is managed, then it was a local cause for d with respect to the previous set of management markings. In the new set h provides no cause for d to become managed, so if there is no other cause, `chunk-unmanage` must be called to mark it unmanaged. The opposite flip can occur when c becomes *un*managed. Without or-chunks, (un)management propagation would be monotonic, and it would obviously converge to a closed set of marks M. Or-chunks make it nonmonotonic, in the sense that marking one chunk can cause another to become unmarked. This raises the possibility of infinite loops.

In fact, it is not hard to construct a network of chunks that does allow infinite loops. Figure 2 shows the simplest case. Here or-chunk h has two disjuncts c and d, with d being the default. d's basis happens to be $\{c\}$. Initially none of the chunks is managed. If `chunk-manage` is called with h as argument, it first sets $M(h) =$ `t`, then $M(d) =$ `t`, then $M(c) =$ `t`. At this point there is no longer any local cause for d, so it becomes unmanaged. Now there's no reason to manage c, so .... In this case there isn't a legal marking. In other networks there are multiple possible markings; in others, various combinations must be tried until a stable labeling can be found.

In all of these examples, the chunk network contains a certain kind of pathological subgraph. A *down link* is a pair of chunks $\langle c_1, c_2 \rangle$ such that c_2 is an immediate supporter of c_1. A *lateral chain* is a sequence $c, h, d_1, d_2, \ldots, d_n$, where h is an or-chunk, d_1 is its default chunk, c is another disjunct of h, and for all i such that $1 \leq i < n$, $\langle d_i, d_{i+1} \rangle$ is a downlink. Note that such a chain is defined independent of any marking of the chunks (cf. supporting paths). The idea is that for some assignment of "managed" or "unmanaged" status to the nodes involved, a change in the management status of c can cause a change in the status of d_1 and hence to d_n.[5] This is called a lateral chain because it allows management marks to flow from a chunk at one height to a chunk at some unpredictable other height. A *lateral cycle* is defined as a series $c_0, \ldots, c_n$ such that $c_n = c_0$ and for all $i \in [0, n-1]$, there is a lateral chain connecting c_i and c_{i+1}. It is fairly straightforward to prove:

> *Theorem:* `manage` and `unmanage` can get into an infinite recursion only if one of them is applied to a chunk c_0 that is part of a lateral cycle $c_0, \ldots, c_n$.

Unfortunately, lack of space prevents me from including the proof here.

This theorem is good news, because it means a simple algorithm can handle all the nonpathological cases, and detect the pathological ones. It is easy to see why lateral cycles are pathological. The purpose of or-chunks is to allow for a default information source to be supplanted by a larger source once there is a reason to load it. In a lateral cycle, each chunk c_i plays the role of "default subset" in the lateral chain to its left, and as "contingent superset" in the lateral chain to its right. It would be unusual to see this pattern carried out for two or three iterations, but downright absurd to see it form a cycle, because the intuitive subset/superset picture would result in a chunk being a superset of itself.

Hence rather than try to develop sophisticated algorithms for coping with lateral cycles, we adopt the much simpler tactic of detecting them and signaling an error. This is easy to do. We simply augment `chunk-manage` with code to set the management state of its argument to `:in-transition`; and augment

[5] Actually, there are chunk graphs in which a given lateral chain can *never* become a conduit in this way, because the required marking is not in fact consistent with the graph's topology. As will shortly become clear, we err on the side of caution by regulating all lateral chains, not just those that are effective.

`chunk-unmanage` with code to check whether the state = `:in-transition`, indicating an attempt to reset before the completion of a set.

3 Updating Chunks

A chunk actually does something useful when it is *updated*, meaning `rederived` if necessary so as to be consistent with its supporters. Exactly how it is determined that some chunks require updating is outside the scope of the chunk system. For instance, consider the (leaf) chunk corresponding to the contents of a source file. Its deriver does not do anything to the contents of the file, but merely changes the chunk's date, if necessary, to equal the write date of the file. There may perhaps be a way to have the file system send a signal to Lisp when the write date changes, but for now I assume that after editing a file the user tells the chunk system to check the new write date and infer the consequences of its having changed.

The program that takes over at this point is called `chunks-update` (plural because in the general case we have a *set* of chunks that have changed). The job of `(chunks-update` *chunks*`)` is to rederive all the *chunks*, but it takes this opportunity to update all the supporters and derivants of the directly affected chunks. (Chunk c_1 is a *derivant* of c_2 if c_2 is a supporter of c_1.) This is a surprisingly complex operation, because of two factors:

1. At the time a chunk is derived, its update basis (p. 3) must be up to date.
2. Updating one chunk may cause the basis of other chunks to change.

Setting these two factors aside for the nonce, the basic algorithm is fairly standard:

```
(defun chunks-update (chunks)
   (let (derive-mark)
      (labels ((chunks-leaves-up-to-date (chunkl)
                  (let ((need-updating '()))
                     (dolist (ch chunkl need-updating)
                        (let ((sl (check-leaves-up-to-date ch)))
                           (setq need-updating
                                 (nconc sl need-updating))))))

               (check-leaves-up-to-date (ch)
                  (chunk-derive-date-and-record ch)
                  (cond ((and (chunk-is-leaf ch)
                              (= (Chunk-date ch) +no-info-date+))
                         (chunk-derive-and-record ch)))
                  (let ((to-be-derived
                              (check-from-derivees ch)))
                     (cond ((chunk-is-leaf ch)
                            to-be-derived)
                           (t
                            (nconc
                               to-be-derived
                               (chunks-leaves-up-to-date
                                            (Chunk-basis ch)))))))

               (check-from-derivees (ch)
                  (let ((updatees
                           (remove-if
                              (lambda (c)
```

```
                        (of (chunk-up-to-date c)
                            (not (Chunk-managed c))))
                     (set-latest-support-date ch))))
            (cons ch
                  (chunks-leaves-up-to-date updatees '())))))

      (derivees-update (ch)
         (cond ((and (Chunk-managed ch)
                     (not (Chunk-derive-in-progress ch))
                     (not (chunk-date-up-to-date ch))
                     ;; Run the deriver when and only when
                     ;; its basis is up to date --
                     (every #'chunk-up-to-date
                            (Chunk-basis ch))
                     (not (chunk-is-marked ch derive-mark)))
                (chunk-mark ch derive-mark)
                (chunk-derive-and-record ch)
                (derivees-derivees-update
                      (Chunk-derivees ch)))))

      (derivees-derivees-update (l)
         (dolist (d l)
            (dolist (c (set-latest-support-date d))
               (derivees-update c)))))
;; BODY OF LABELS BEGINS HERE
(setq derive-mark chunk-event-num*)
(setq chunk-event-num* (+ chunk-event-num* 1))
(let ((chunks-needing-update
          (chunks-leaves-up-to-date chunks '())))
    (dolist (ch chunks-needing-update)
       (derivees-update ch '()))))))
```

The final version of the algorithm is much more complex than this, but we can already discern some subtleties. The function `chunks-update` calls a few important subroutines:

- `(set-latest-support-date` c`)`: Compute the date of the most recently updated base of chunk c. If it is later than c's `latest-support-date`, reset that slot of c, and repeat the computation for each derivee of c. Returns a list of all the derivants of c the date of whose most recently changed supporter has changed.

- `(chunk-derive-and-record` c`)`: Apply `derive` to c, and change c's date to the date returned by `derive` if it is later than c's old date. While this is going on, set the `derive-in-progress` slot of c to `t`.

- `(chunk-derive-date-and-record` c`)`: Apply `derive-date` to c, and set c's date accordingly. Returns `t` if the new date is newer than c's old date.

- `(chunk-mark` c m`)` and `(chunk-is-marked` c m`)`: See below.

The `chunks-update` algorithm proceeds in two phases[6]: During the outer call to `chunks-leaves-up-to-date`, it sweeps through the chunk network finding all the "questionable" chunks reachable from the given list. A chunk is *questionable* if it is managed, it is out of date, and either it is in the list `chunks`, or it supports a questionable chunk, or it is derived from a questionable chunk. The out-of-dateness test is not

[6]For brevity, I've omitted the — entirely straightforward — code that checks for support cycles during the sweeps through the chunk network.

performed using the dates pasted on chunks when the sweep starts, but on the dates that emerge when leaves supporting questionable chunks are (re-)derived.

This sweep returns a list of non-leaf questionable chunks. In the second phase, the outer call to `derivees-update`, these chunks are re-derived. Note that `derivees-update` simply discards a chunk whose basis is not up to date. That's because the questionable basis chunks must themselves be on the list of chunks to try deriving; when `derivees-update` gets to them, and re-derives them, it will then call itself recursively to derive their derivees. Eventually all questionable derivees will be updated, except for the rare cases in which `derive` determines that a questionable basis is up to date after all.

The algorithm uses a marking scheme to ensure that no chunk is derived more than once. The global fixnum `chunk-event-num*` is stored as `derive-mark`, and then incremented so that the same number is never used again. Whenever `derive` is applied to a chunk, the chunk's `update-marks` is field is used to record that it is now marked with `derive-mark`. The function `chunk-is-marked` checks to see whether the chunk has already been marked with `derive-mark`. If the report is positive, then the chunk is not derived again. Through all the complexities that are added to `chunks-update`, this property is preserved, because no matter how the chunk network changes, the deriver of a chunk is supposed to take all relevant data into account when it runs.

Lack of space prevents me from a thorough description of the actual `chunks-update` program. The following is a very skimpy sketch of the layers of complexity that must be added to the basic code above.

The update basis of a chunk must be up to date before the chunk is derived. This requires a change to `derivees-update`. However, before control gets to that point, the update basis must be managed, or its components will not be updated. We must add code to `check-leaves-up-to-date` to call `chunk-manage` on the update basis of a chunk that might be updated. All such temporarily managed chunks are placed on a list, and when `chunks-update` is finished, it calls `chunk-unmanage` on them. This code is `unwind-protect`ed so that the temporary management is undone even if `chunks-update` terminates in some abnormal way.

In addition to the derivation mark, the chunk-update system must use different marks to mark chunks that have been seen during `chunks-leaves-up-to-date` and those seen during `derivees-update`. The same global counter, `chunk-event-num*`, is used for this purpose, and we call the two marks `down-mark` and `up-mark` respectively. We do *not* provide three slots on each chunk to keep track of these marks, because of the possibility of unexpected calls to `chunks-update`, a topic to which I now turn.

As I have mentioned more than once, there is no way to keep chunk derivers from calling `chunks-update`. When it happens, we must let the call proceed, because it may change the outcome of the current call. For instance, one system of chunks may keep track of which files depend on which other files, while another keeps track of the compilation and load states of files. During an update of the latter system, an update of the former may occur, thus changing *which* files should be be compiled or loaded. We can say informally that the first system is "meta" to the second, but I've made no input to introduce explicit "layers" and "metalayers" to the chunk system. Instead, when a call to `chunks-update` detects that another call has happened, it simply restarts.

Restarting means allocating new values for `down-mark` and `up-mark`, then marking from `chunks` all over again. The way marks are managed is that each chunk has a *list* of marks. To tell if a chunk is marked with m, the system checks to see if m is in the list. Rather than use `member`, as it traverses the list it deletes marks that are no longer in use. To tell if a mark is still in use requires `chunks-update` and other "mark-allocating" functions to *discard* marks they have allocated; this occurs when `chunks-update` exits, normally or abnormally.

Although the details of this scheme are entirely orthogonal to chunk management, it does give us an easy way of testing whether `chunks-update` has been called by someone while `chunks-update` was in progress: simply check to see if some other process has allocated a chunk mark. When this event is detected, `chunks-update` drops what it is doing, and restarts.

4 Applications and Conclusions

The biggest application of the chunk system is the YTools File Manager (YTFM) [3], but is impossible to talk about all its intricacies in the space available. Besides, a simple example will show better how much value is added by using chunks.

Let's suppose that a file `tab.lisp` initializes a table with some sort of S-expression handlers, each associated with a symbol that can occur as the `car` of an S-expression. In `tab.lisp` we can have this code:

```
(declare-chunk handler-table-init
   :contents
      ((defparameter handler-table* (make-hash-table ...))))
```

In a later file `handlers.lisp` we can write

```
(declare-chunk special-form-handlers (:depends-on handler-table-init)
   :contents
      ((setf (gethash 'cond handler-table*)
             (lambda (x y z) ...))
       (setf (gethash 'let handler-table*)
             (lambda (x y z) ...))))
```

To make the `declare-chunk` macro work, all we need to do is define a class `File-segment-chunk`, which has two kinds of base chunk: the `File-chunk` of the file the chunk declaration appears in, and the chunks it is declared to depend on. The `File-chunk` abstraction is supplied by the YTFM, as is the closely related `Loaded-file-chunk`, which manages "File F is loaded into memory." We need the latter for the update basis of a `File-segment-chunk`; to update a chunk declared in file F, it is necessary (and sufficient!) for F to be loaded. The `contents` of a `File-segment-chunk` become a function with zero arguments, to be called by `derive` when applied to an element of the class. In the example, if `tab.lisp` is reloaded, then if `handlers.lisp` hasn't changed since it was last loaded, then `derive` calls the function, thus re-evaluating the two `setf`s. If `handlers.lisp` has changed, then the deriver does nothing (because the file will have been reloaded before the deriver is called). This is the simplest scheme, but it is easy to explore other alternatives, such as "slurping" the file to find and evaluate just the chunk definition.

The point is that this mechanism allows fine-grained control over the rebuilding of Lisp sessions. Once the dependency has been declared, the developer can stop worrying about it, confident that the chunk manager will always reconfigure data structures properly as files are debugged and reloaded. With this confidence, the times when the user must give up and reload everything can be reduced to a minimum.

References

[1] Jim Farley. *Java Distributed Computing.* O'Reilly, 1998.

[2] Heiko Kirschke. *Persistent Lisp Objects!* At `http://plob.sourceforge.net/plob.html`, 2005.

[3] Drew McDermott. *YTools: A Package of Portable Enhancements to Common Lisp* . Available at `http://cs-www.cs.yale.edu/homes/dvm/papers/ytdoc.pdf`, 2005.

[4] Kent Pitman. The Description of Large Systems. Technical Report 801, MIT AI, 1984. Now available at `http://www.nhplace.com/kent/Papers/Large-Systems.html`.

[5] Rosenberg. ASDF:, 2004. Another System Definition Facility. `http://www.cliki.net/asdf`.

[6] Kenny Tilton. *Cells: A Dataflow Extension to CLOS.* `http://common-lisp.net/project/cells/`, 2005.