Chapter 9

Experience

This chapter describes the experimental aspects of my research. I discuss the systems I have implemented to experiment with my ideas, describe the tests I have undertaken with these systems, and evaluate the current status of my work based on this experience. I also summarize a few of the important lessons that I learned in the context of this research.

9.1 Implementation Notes

As a part of my research in reducing the tension between modularity and control, I have constructed prototype implementations of the EDGAR, OPERA, and SYNAPSE languages described in the previous three chapters. Additionally, I have implemented a graphical program animator, the DYNAMATOR, that has proven invaluable for guiding the design of these languages and debugging their implementations. This section presents a synopsis of each of these four systems.

9.1.1 EDGAR

All of the languages and tools mentioned above are based on the EDGAR interpreter, which implements the graph rewrite rules described in Chapter 8. Currently, the EDGAR interpreter is a Scheme program that supports all the EDGAR features except for vectors; these should be easy to add.

Basics of the EDGAR Interpreter

The EDGAR interpreter represents graph graph nodes and wires as Scheme objects connected in a way that directly reflects the topology of the graph being modelled. Each node object maintains a list of input wires, a list of output wires, and node-dependent state information. Each wire object maintains its source node, its target node, and wire-dependent state information. There is a library of Scheme procedures for creating nodes and wires, hooking them together, and accessing and modifying their states.

Graph rewriting is handled by mutating a global graph structure via an agenda-based simulation. An *agenda* is a list of *tasks* to be peformed. Sample tasks include propagating requests from an output wire of a node to its input wires, returning a value to a requested wire, and creating new nodes that are inserted into the graph. Performing one task on the agenda may add other tasks to the agenda or remove other tasks from it. The EDGAR interpreter is invoked with a initial graph and an initial list of tasks for the agenda. It performs one task at a time until there are no more tasks on the agenda. The "result" of the EDGAR interpreter is the graph structure remaining when there are no more tasks to be performed.

The basic architecture of the EDGAR interpreter was inspired by Abelson and Sussman's digital logic simulator [ASS85]. The representation of networks, the propagation of information, and the agenda-based simulation closely resemble aspects of the digital logic simulator. The main innovation of the EDGAR interpreter is the dynamic nature of its networks. While logic networks are static, the structure of EDGAR graphs changes over time with the addition and removal of nodes. In this respect, the EDGAR interpreter can profitably be viewed as a cross between the digital logic simulator and a graph reduction engine.

Task Selection Strategies

Due to the nondeterminism of EDGAR's rewrite rules, there may be more than one task to perform at a given time. After completing one task, the EDGAR interpreter employs a *task selection strategy* to choose the next task on the agenda. The default strategy is to randomly select the next task from the agenda. But EDGAR comes equipped with a variety of other strategies that aid experimenting and debugging. For example, it is possible to prioritize the tasks, assign probabilities to them, or choose them manually. Another option is to "play back" a list of choices constructed by a previous invocaton of the EDGAR interpreter; this facilitates debugging in the presence of the random selection strategy.

In the current implementation of EDGAR, the graph rewrite rules are not localized anywhere. Instead, they are distributed over the behaviors of the individual node types. The procedural, distributed nature of the rules makes them difficult to locate, understand, modify, and prove correct. To overcome these difficulties, it would be worthwhile to design a separate language for expressing the set of graph rewrite rules in a clear and concise way.

Garbage Collection

Perhaps the most interesting technical detail of the EDGAR interpreter involves garbage collection. Recall that the EDGAR rewrite rules may create nodes and wires and move wires but they do not delete nodes or wires. Instead, it is assumed that a garbage collection process removes all nodes and wires that are inaccessible from a root node (i.e., a sink node or an eagon node).

In EDGAR, garbage collection is not only a practical matter, but a semantically important one: the proper behavior of synchrons depends on it. The rendezvous rule for synchrons dictates that a rendezvous occurs at a synchron when it is referenced only by waiting pointers — i.e., when all the output wires of a synchron are in the returned state and are connected to requested wait nodes. Since any non-waiting pointer blocks a rendezvous, it is essential to eventually remove these wires when they cannot be accessed from a root. Otherwise, it is possible to enter a deadlock state in which all forward progress in a computation is impeded by a spuriously blocked rendezvous. Even if an EDGAR interpreter were magically given an infinite amount of memory in which to execute, it would still need to perform some sort of storage analysis to effectively reclaim inaccessible non-waiting wires.

Let's ignore the space reuse aspect of garbage collection in EDGAR and instead focus on the synchronization aspect. A naive approach to avoiding spurious deadlocks is to perform a garbage collection whenever a deadlock occurs, and then examine all synchrons to see if the garbage collection has enabled a rendezvous. If so, the computation can continue with that rendezvous; otherwise, the computation halts in a true deadlock state. While this approach is correct, it is not necessarily practical in a system where synchronization is frequent and garbage collection is potentially expensive. Another drawback of this scheme is that it impairs debuggability of EDGAR graphs by hiding the accessible nodes in a sea of inaccessible ones.

An alternate approach is to use a reference counting garbage collector that *drops* (immediately removes) nodes and wires as soon as they become inaccessible. Moving or dropping an output wire of a node triggers a check that determines whether the node has any output wires remaining. If not, it can safely be dropped, which entails dropping its input wires as well. The recursive nature of dropping means that a single rewrite rule can trigger the dropping of an arbitrarily large subgraph. As typical with reference counting schemes, this approach does not interact well with cyclic structures. It is possible for a synchron to be spuriously held by an inaccessible cyclic data structure in such a way that inaccessible non-waiting pointers to the synchron are never dropped by the reference counting scheme. However, such situations tend to be rare in practice.

The EDGAR interpreter combines the reference counting approach with the naive approach. Reference counting is applied as a default, but a full-fledged garbage collection is invoked whenever a deadlock state is reached. In the presence of reference counting, a deadlock state almost always represents a program error rather than a spuriously blocked rendezvous, so the price of a full-fledged garbage collection is rarely paid by most programs. Moreover, under this approach, the current set of undropped nodes and wires is usually a very close approximation to the set of accessible nodes and wires; this greatly simplifies the debugging of EDGAR graphs.

Global Cells

In order to support the interactive evaluation of individual OPERA expressions and definitions, the EDGAR interpreter maintains a global table mapping top-level names to cells. These global cells facilitate the sharing of values (especially procedures) between one computation and the next. Global cells act as an additional type of root node for garbage collection.

Metering

The EDGAR interpreter supports metering operations that keep track of the following parameters of a computation:

1. The number of tasks performed during a computation.

2. The number of nodes created during the computation.

3. The maximum number of nodes created during the computation that are in use at any intermediate point of the computation.

4. The number of nodes accessible at the end of a computation.

5. The process time required by a computation.

6. The real (wall clock) time required by a computation.

These quantities are useful for comparing the time and space behaviors of different programs.[1]

Debugging

The EDGAR environment supports a number of debugging tools. Without a doubt, the most important of these is the DYNAMATOR, a graphical program animator discussed in Section 9.1.4. But there are some other useful features, including the ability to single step through a program, trace procedures, and set breakpoints at procedure entry or synchron rendezvous.

Liberties

For reasons of simplicity and efficiency, the prototype EDGAR implementation takes a few liberties with the EDGAR rewrite rules:

[1]According to the definition of space requirements in Section 8.1.5, space includes not only the number of nodes but the number of wires. The current version of the EDGAR interpreter does not keep track of the number of wires; this should be changed in future versions. There are some pathological cases involving synchrons where it's important to account for the number of wires, but these don't crop up in any of the examples I considered.

- *Aggressive returns*: According to the conventions of the EDGAR rewrite rules, a wire is not supposed to enter a returned state until it has been requested. In practice, this means that a large number of rule applications are devoted to requesting and returning "obvious" values like numbers and lazons. To avoid these rule applications, the EDGAR interpreter immediately "returns" such values to their output wires without waiting for a request. Other rules are modified to take into account the possibility of these unrequested returns.

- *Implicit touches*: The EDGAR rewrite rules require explicit `touch` nodes to require the results of a computation associated with an eagon or lazon value. In order to reduce the number of touch nodes and the number of touching rules applied, the current implementation implicitly touches nodes in the following contexts:

  - the arguments of a strict primitive procedure call.
  - the test input of an `if` node.
  - the operator position of a `pcall` node.
  - the lock position of an `exclusive` node.

- *Sysprim nodes*: According to the presentation in Section 8.1, primitive applications in EDGAR are expressed via distinguished primitive application nodes. While the EDGAR interpreter supports primitive application nodes, it also introduces literal nodes called *sysprim nodes* that represent primitive procedures. A `pcall` node whose operator is a sysprim node is treated exactly like a primitive application node.

- *Alternate evaluation strategies* EDGAR permits experimentation with various procedure application strategies by supporting evaluation strategies other than the default concurrent one. In particular, it also supports strict sequential strategies (both ordered and unordered), a lazy strategy, and an eager strategy. For example, the unordered strict sequential strategy is used to implement a "Scheme mode" that facilitates the evaluation of Scheme (as opposed to OPERA) programs.

9.1.2 OPERA

The OPERA language outlined in Chapter 7 has been implemented as an evaluator written in Scheme. The current version of OPERA handles most standard Scheme features. The Scheme features currently not handled are: vectors, continuations, rest arguments, `apply`, and multiple-value returns. Eventually, I plan to add all of these to OPERA. The handling of continuations will be based largely on Bawden's work (see [Baw92]).

The OPERA evaluator works by compiling OPERA expressions into EDGAR graphs that are interpreted by the EDGAR interpreter (see Section 8.3 for details). The EDGAR graph representing an OPERA expression is rooted at a top-level `sink` node to provide the initial "tug" for evaluating the graph in a demand driven fashion. Evaluation of an OPERA expression terminates only if the EDGAR computation terminates (i.e., the agenda becomes empty). At the termination of a computation, the input wire of the top-level `sink` node is examined for one of two cases:

1. If the wire is in the returned state, then the subgraph rooted at the source of the wire represents the value returned by the computation. This subgraph is translated into a format appropriate to OPERA and returned as the value of the initial OPERA expression.

2. If the input wire to the top-level `sink` node is in the requested state, then the computation terminated without returning a value. This indicates a deadlock state, which is reported to the user along with an option to debug the deadlock (via the DYNAMATOR).

Note that a computation does not necessarily terminate when a value is returned to the top-level `sink` node. Due to the presence of eagons, there may be other tasks to perform even after a value appears at the `sink` node. All tasks associated with eagons must be performed before a value can be returned to the OPERA evaluator. The rationale for this decision is that the eagon tasks might perform side effects that affect the evaluation of future OPERA expressions; forcing value returns to coincide with termination ensures that top-level OPERA expressions appear to be evaluated atomically.

The interface to the OPERA evaluator is a read-eval-print loop. The read-eval-print loop handles `define` and `load` forms specially. A top-level `define` form compiles into an EDGAR graph that updates the value of a global cell. A `load` form evaluates each of the forms in a specified file as if they were typed in at top level.

The OPERA to EDGAR compiler departs in a number of ways from the description in Section 8.3. The transformation of OPERA to o is does not exhibit all the structures or the features of the modular approach explained earlier. `Quote` is currently handled in an ad hoc fashion that does not faithfully preserve its sharing properties. The template produced by the compilation of a `lambda` expression is actually a delayed compilation of the body.

9.1.3 SYNAPSE

The SYNAPSE language introduced in Chapter 6 has been implemented as a collection of OPERA procedures. Section 7.2 gives an overview of this implementation ([Tur94] presents all of the code for these procedures). While filtered synquences are handled in a fully reusable manner, filtered synquences are currently handled in an ad hoc fashion.

9.1.4 The DYNAMATOR

The DYNAMATOR is a graphical program animator that dynamically animates the evolution of an EDGAR computation. It has been invaluable for guiding the designs of and debugging the implementations of EDGAR, OPERA, and SYNAPSE. It has also helped me build intuitions about concurrency, synchronization and non-strictness in a way that would not have otherwise been possible.

Snaphsot Representation

The dynamator displays a "movie" of computational snapshots, where each snapshot displays the nodes and wires of the EDGAR interpreter's current graph. Each snapshot resembles the manually drawn figures in this document: nodes appear as labelled boxes with triangular input and output ports, and wires appear as lines between ports.[2] Color is used

[2] At the time of this writing, I do not have a convenient way to include a sample DYNAMATOR snapshot in this document.

in place of empty and filled circles to encode the state of a wire (inactive, requested, returned). In DYNAMATOR snapshots, I have also found it worthwhile to define a *node state* that can also be encoded by color. The state of a node is defined as:

- *Returned* if at least one of the output wires is in the returned state.
- *Requested* if some of its output wires are in the requested state, but none are in the returned state.
- *Inactive* if all of its output wires are in the inactive state.

Distinguishing the node states by different visual cues makes it easy to parse a snapshot into values (returned subgraphs), pending operations (requested subgraphs), and code that has not yet been interpreted

Some nodes in a snapshot are distinguished by an indication that they are *enabled*. Recall from Chapter 8 that every potential match of a rewrite rule is associated with an enabled node. In a DYNAMATOR snapshot, every node marked as enabled represents one of the tasks that is on the agenda of the EDGAR interpreter.

Dynamics

The EDGAR interpreter generates *display events* whenever a displayable action happens (e.g., a node or wire is created, a wire changes state, a wire is moved, a node or wire is dropped.) The DYNAMATOR responds to these events by updating the currently displayed snapshot. The result is that every change in the state of the interpreter is reflected by a corresponding change in the display state of the DYNAMATOR. This makes it possible to watch computations dynamically unfold over time.

Since the DYNAMATOR reflects the state of the EDGAR interpreter, the animation of the DYNAMATOR can be controlled by changing the mode of the EDGAR interpreter. For example, under the default processing of the EDGAR interpreter, the animation unfolds as fast as the graphics operations generated by the interpreter can be executed. When the EDGAR interpreter is in single-stepping mode, the animation can be moved forward one snapshot at a time. One of the task selection strategies supported by the EDGAR interpreter allows the user to select an enabled node in the displayed snapshot to indicate which task should be performed next. This gives rises to a single-stepping mode in which the user controls exactly how the computation evolves.

Some display transitions are smoother than others. For example, changing the state of a wire does not require repositioning any of the nodes or wires. Moving a wire requires minimal changes: erasing one line and drawing another. On the other hand, creating new nodes and wires (as is done at every procedure invocation) may require non-trivial reorganization of the visual representations in the snapshot. Currently, the default is to redisplay the entire snapshot at every procedure call. This leads to a "jerkiness" in the visual representation that can make it hard to track individual nodes and wires. It would be worthwhile to investigate strategies to reduce this jerkiness.

Graph Layout

I have experimented with two strategies for laying out the snapshot graph:

- In *graph mode*, a directed spanning tree of the graph is determined from the top-level `sink` node.[3] The node of the spanning tree are laid out in a recursive fashion such that every node appears above its subtrees. Wires that are not part of the spanning tree are then added on top of this basic tree structure.

- In *Escher mode*, all non-call nodes apear at the top of the representation, and the rest of the display space is partitioned among the call nodes, which are represented as large rectangles. For each call, the new node and wires structure created by the call appears in the appropriate rectangle. This approach is called "Escher mode" because the representations get smaller and smaller as the computation progresses.

Each of these approaches has advantages and disadvantages. Graph mode only depends on the current state of the EDGAR interpreter, so it can be invoked at any time during the computation. On the other hand, Escher mode represents the entire call structure of the computation, so it only works for displaying an entire computation from start to finish.

[3]When there is more than one root (i.e., there are unheld `eagon` nodes), they are processed in sequence, and each is considered to be the root of a tree consisting of the nodes that are not yet part of any other tree.

Escher mode has the further disadvantage that the representations quickly get too small to be helpful (some magnification facility would be helpful here!). The key advantage of Escher mode is that a every node has a never-changing position on the display canvas. In particular, when a procedure is called, new nodes are created, but old nodes do not move. In contrast, a procedure call in graph mode currently triggers a redisplay of the entire computation graph. It is worthwhile to see if some other graph layout mechanism could somehow combine the best of both approaches.

Additional Features

Additional features supported by the current implementation of the DYNAMATOR include:

- *Code view*: Nodes that resulted from compiling OPERA code are annotated with the OPERA expression responsible for their generation. Procedure nodes are also annotated with the code of their bodies. These annotations can be accessed by selecting the visual representation of the node.

- *Procedure names*: A disadvantage of the process of compiling OPERA to EDGAR is that the names of the program are lost. But names carry a significant amount of information. The current implementation keeps track of the names associated with procedures and uses these names in place of the usual `proc` label for procedure nodes.

- *Abbreviations*: When snapshots contain large numbers of nodes, the individual nodes are so small that long labels are impractical. The DYNAMATOR makes it possible to abbreviate the labels so that such snapshots can still be understood.

- *Dropped images*: When nodes or wires are dropped, there is an option to completely erase them from the screen, as well as an option to show them in a muted gray. The latter option makes it possible to tell something about the recent history of the current snapshot.

- *Node highlighting*: Nodes and wires can be programatically highlighted in a chosen color. This facilitates finding which node is responsible for an error when debugging.

In the current implementation, the interface to many of these features is programmatic rather than interactive. The reason for this is that the version of Scheme I use has a poor interface to the X window system. I plan to explore better interfaces to this functionality.

The kinds of features listed above only scratch the surface of what should be provided in a full-fledged program animator. Other desirable features include: starting and stopping a computation at the press of a button; running a computation backwards (at least for some predetermined number of steps); manually dragging nodes across the screen to improve upon automatic layout; highlighting subgraphs; selecting and expanding subgraphs; annotating wires with names determined by the program; and graphically editing a program graph.

9.2 Testing

I have touted the sliver technique as a mechanism that facilitates the modular expression of some computations while retaining their operational character. I have also claimed that, for certain programs, the sliver technique is superior to other modularity mechanisms in certain dimensions. While I can sketch informal arguments that slivers provide the claimed guarantees in specific cases, I do not have any formal proof of these claims. What I *do* have is extensive experience interacting with the EDGAR, OPERA, and SYNAPSE implementations. Here I summarize the kinds of tests I have performed with these systems and explain how they support my claims.

The goal of the tests is to show that SYNAPSE programs exhibit the operational behavior that they were designed to exhibit. The notion of operational behavior could be formalized in terms of the behavior of an EDGAR initial graph — i.e., as a set of computation traces. However, given that the set of traces can be very large even for a small program, I believe that it is impractical to develop a testing strategy based on this formalism. Instead, I focus on three testable characteristics of this formalism for behavior: the *outcome* of a program, individual *computations* of a program, and the *space requirements* of a program.

9.2.1 Outcomes

The very least that we expect from a SYNAPSE program is that it has the expected outcome. Recall that there are three classes of outcomes in the underlying EDGAR model: a program can either return a value, enter a deadlock state, or hang in an infinite loop.[4] Although there is no effective test for the looping outcome, it is easy to test for returned values and deadlock.

I have implemented an automatic testing program that verifies that an OPERA expression has a specified value/deadlock outcome. The testing program runs successfully on a suite of hundreds of simple SYNAPSE programs. Most of the programs are similar to the kinds of synquence and syndrite examples illustrated in Chapter 6. Many of the test programs are examples that failed to work in earlier incarnations of the system.

The most important aspect of automatic testing is that it can rapidly uncover spurious deadlocks. Programs exhibiting shape incompatibilities are supposed to deadlock. However, if the SYNAPSE primitives are not implemented extremely carefully, it is possible for programs to deadlock when they are supposed to return a value. To aid in discovering these bugs for syndrite slivers, I wrote a program that generates test programs for all possible pairings of syndrite accumulators and scanners. This automatic test generator embodies a simple notion of shape compatibility that accurately predicts which combinations should deadlock.

9.2.2 Computations

Recall that a computation for a program is a sequence of EDGAR graphs comprising one of its possible executions. Using the DYNAMATOR, it is possible to watch an entire computation unfold from start to finish. Textual operation traces, like those employed throughout Chapter 6, also provide a good window onto a computation.

I have not developed any tools for automatically analyzing computations, but I have had studied hundreds (if not thousands) of DYNAMATOR animations and operation traces. Based

[4] Due to the nondeterminism inherent in the systems, it is possible for single program to return multiple values and/or exhibit all three kinds of outcomes in different executions. However, we will only consider programs that have a single outcome.

on this experience, I have great confidence that SYNAPSE programs exhibit the predicted operation order. In particular, sliver operations are always performed between the right pair of synchron rendezvous.

Given the nondeterminism present in OPERA programs, it is reasonable to wonder how much faith should be invested in particular animations and operation traces. After all, it may be that the expected operation order is highly probable, but not guaranteed. Nevertheless, there are two reasons to expect that the computations I examined accurately characterize the overall behavior of the tested programs:

1. *Diabolical operation scheduling*: In many cases, I executed the programs using a worst-case operation scheduling strategy. The EDGAR interpreter has a hook for installing an arbitrary strategy for determining which enabled rewrite rule to apply next. I have implemented a higher-order strategy generator that makes it possible to specify the relative priorities of different operations.

   The priority strategy can be used to force a program to exhibit possible (but unlikely) undesirable behavior. For example, consider the following SYNAPSE expression:

   ```
   (let ((nums (to-1 5)))
     (cons (downQ 0 + nums)
           (downQ 1 * nums)))
   ```

   In a correct implementation, the accumulation operations should always occur in pairs of one + and one *. Yet, even a concurrent lazy implementation of `downQ` is likely to exhibit this behavior. To distinguish between the two implementations, we can set the priority of + below all other operations (including *). In the concurrent lazy implementation, this will force all *s to precede the first +, In a correct SYNAPSE implementation, the * and + operations will still occur in pairs, but the * will precede the + in every pair.

2. *"Funneling" behavior of synchrons*: Although the nondeterminism present in OPERA programs can lead to a blow-up in the number of possible computations, the use of synchrons greatly constrains the nature of these computations. Synchronized lazy aggregates were specifically designed to force concurrently executing slivers to engage

in a barrier synchronization. At the time of a rendezvous, every sliver should be in a state where only the rewrite rule for a rendezvous is enabled. Although there are many possible paths that a SYNAPSE program can take from one rendezvous to the next, the synchronization guarantees that computations can never wander too far away from a sample path.

Extensive experience watching DYNAMATOR animations confirms that synchrons regularly "funnel" SYNAPSE computations to a single point before allowing them to proceed. Inspecting the dependencies among operations and `wait`s before and after a rendezvous is the basis for "visual proofs" that the operations must execute in the desired order.

9.2.3 Space Requirements

Perhaps the most important characteristic of a SYNAPSE program is its space behavior. After all, a major motivation for slivers was to design an aggregate data model of computation in which the space required by the aggregates could be controlled in a reasonable way. There are two aspects to space behavior:

1. *Order of growth*: How does the space requirement of a program depend on the size of the input?

2. *Space profile*: How does the space consumed by an individual computation change over time?

The goal of operational faithfulness dictates that a SYNAPSE program should qualitatively match these space characteristics of the corresponding monolithic loop or recursion.

The EDGAR interpreter provides mechanisms for tracking the space profile and maximum space usage of a computation.[5] These were used in experiments that compared the space behaviors of SYNAPSE programs to those of monolothic programs and other aggregate data programs. We describe two such experiments here: a linear example (averaging the numbers of a file) and a tree example (alpha renaming).

[5]The current implementation models the space of an EDGAR graph as the number of nodes in the graph. According to the definition of space in Section 8.1.5, wires should also be accounted for. However, including wires should not change the qualitative nature of the results discussed below.

Average

The averaging program finds the average of the contents of a specified numeric file. Figure 9.1 shows monolithic and modular versions of this program written in OPERA. These

```
;;; Monolithic version

(define (average-file-down:mono filename)
  (let ((port (open-input-file filename)))
    (define (loop sum count)
      (let ((elt (read port)))
        (if (eof-object? elt)
            (begin (close-input-port port)
                   (/ sum count))
            (loop (+ elt sum) (1+ count)))))
    (loop 0 0)))


;;; Modular version

(define (average-file-down:mod filename)
  (let ((nums (splay-file filename)))
    (/ (downQ 0 + nums)
       (downQ 0 inc nums))))

(define (inc elt sum) (1+ sum))

(define (splay-file filename)
  (let ((port (open-input-file filename)))
    (produceQ 'ignore
              (lambda (ignore yield terminate)
                (let ((elt (read port)))
                  (if (eof-object? elt)
                      (begin (close-input-port port)
                             (terminate))
                      (yield elt 'ignore))))))))
```

Figure 9.1: Monolithic and modular versions of an iterative program for averaging the contents of a numeric file.

two programs were tested under conditions that ranged along the following dimensions:

- *Sequential vs. Concurrent*: The default application strategy of the EDGAR interpreter is to concurrently evaluate the subexpressions of all procedure calls. To facilitate comparing sequential and concurrent evaluation strategies, the EDGAR interpreter also provides a sequential mode in which the subexpressions of all calls are evaluated in some sequential order.

- *Strict vs. Lazy vs.* SYNAPSE: The default implementation of the SYNAPSE sliver primitives is based on synchronized lazy aggregates. For the purposes of comparing different aggregate data approaches, I also implemented versions of the SYNAPSE primitives based on strict aggregates and lazy aggregates.[6]

- *Random vs. Prioritized*: To test likely vs. worst-case scenarios, the averaging programs were run under different task selection strategies. The random strategy picks the next task randomly. The prioritized strategy gave `+` a lower priority than all other operations (including the incrementer, `1+`).

To test the averaging programs for order of growth in space, their space requirements were measured as a function of file size. The highlights of the results appear in Figure 9.2. The figure shows an overlay of plots for six experiments. The labels for the individual plots indicate the conditions under which they were run. (Those plots with a "prioritized" label were run under the prioritized strategy, while those without this label were run under a random strategy.) The figure shows the following results:

- Even under worst case scheduling conditions, the modular SYNAPSE program exhibits a constant space requirement that qualitatively matches that of the monolithic program. Quantitatively, the SYNAPSE program exhibits a higher constant factor than the monolithic program.

- Sequential strict and sequential lazy implementations require space linear in the size of the file.

- Under favorable conditions, a concurrent lazy implementation can exhibit constant space requirements. However, under worst-case conditions, a concurrent lazy implementation requires space linear in the size of the file.

[6] There is the potential problem that my strict and lazy implementations do not make optimal use of space. It would have been better to test the programs in "industrial strength" implementations of established languages that support strict and lazy features, but it was impractical to do so.

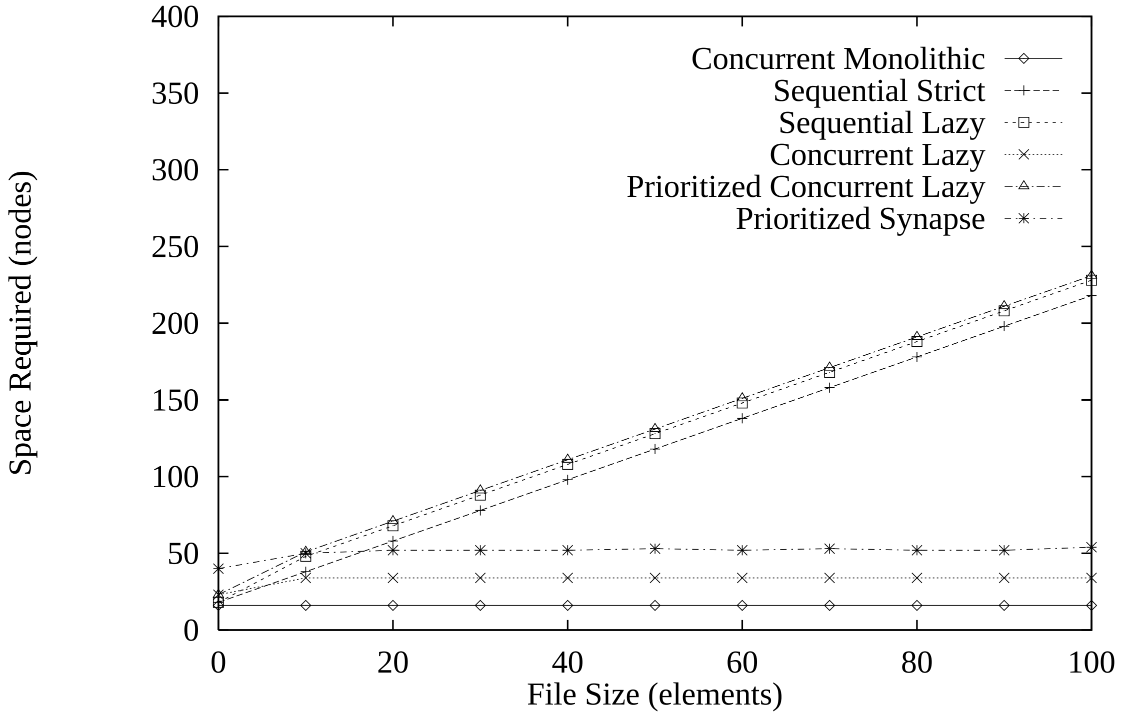


Figure 9.2: Comparison of the space requirements of various averaging programs as a function of file size.

To compare space profiles of individual computations, sample space profiles were generated by running the averaging programs on an input file with ten numbers. Profiles for six different strategies appear in Figure 9.3. The x-axis measures time in in execution steps of the EDGAR interpreter, while the y-axis measures space in the number of nodes required by the EDGAR graph at the given execution step. The space profiles can be classified into three qualitative categories:

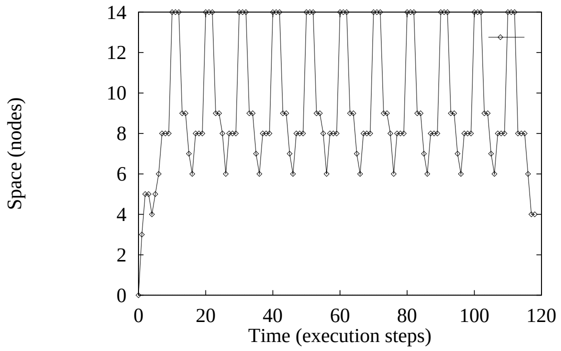
1. *Flat*: The three constant-space computations (concurrent monolithic, concurrent lazy, and prioritized SYNAPSE) all exhibit a broad flat band in which the space usage alternates between high and low values ten times. Each high value is the size of the graph soon after the beginning of a new iteration, while each low value is the size of a graph at the end of an iteration. Irregularities in the high and low values for the modular programs are due to the nondeterminism of the concurrent evaluation strategy. The peaks do not indicate a rising trend in the two constant-space modular versions because the intermediate values held by the aggregates are aggressively dropped rather than being buffered.

   Due to management operations, the modular versions take significantly more steps than the monolithic version. In the case of the SYNAPSE program, the difference is an order of magnitude.[7] The management operations for the SYNAPSE program include packaging and unpackaging slags, synchronzing on synchrons, and handling higher-order procedural arguments.

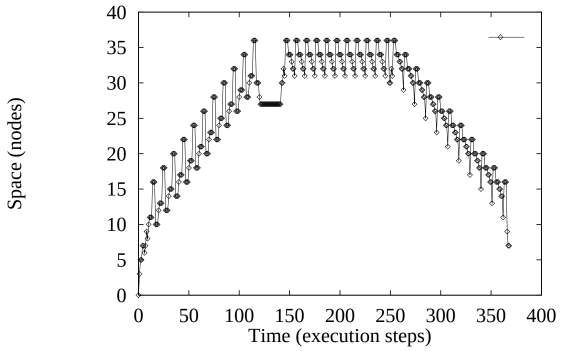
2. *Peak*: The sequential lazy and prioritized concurrent lazy programs exhibit profiles that gradually rise to a peak and then fall off more rapidly. The rise is due to the fact that each element read from the file is buffered for later use; the fall occurs in the phase of the computation that processes the buffered elements.

3. *Plateau*: The sequential strict program exhibits a plateau between a rising segement and a falling segment. Each of these three segments corresponds to one of the three

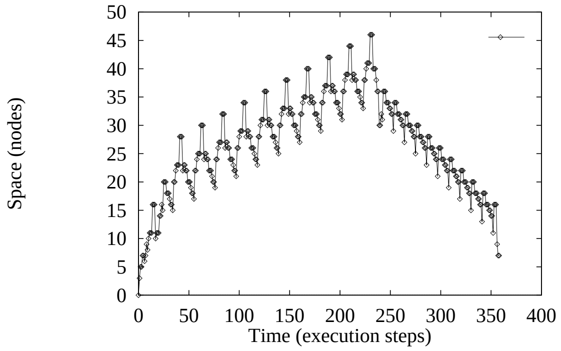
[7]The SYNAPSE profile was generated in an unfiltered implementation of SYNAPSE The time overhead would be even higher in the filtered version.



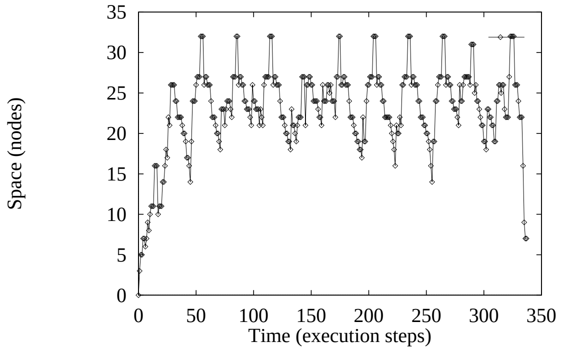
(a) Concurrent monolothic



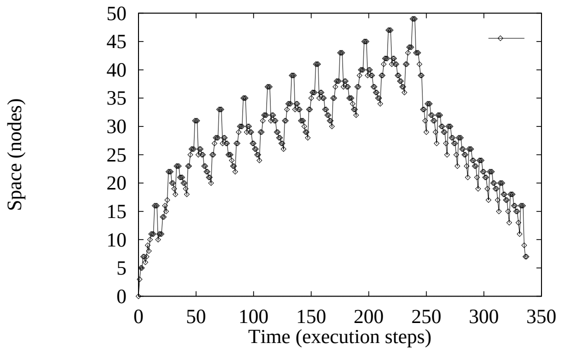
(b) Sequential strict



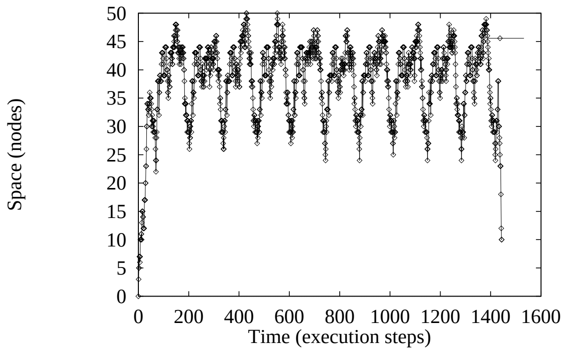
(c) Sequential lazy



(d) Concurrent lazy



(e) Prioritized concurrent lazy



(f) Prioritized SYNAPSE

Figure 9.3: Space profiles of various *down* averaging programs on an input file of ten elements.

list-manipulation modules. The rising segment is caused by `splay-list`'s construction of a list holding all the elements of the file.[8] A plateau occurs during the first accumulation because all of the list elements must be saved for the second accumulation. During the second accumulation, the list elements are dropped as soon as they are processed.

Experiments were also performed on *up* versions of the file averaging programs (Figure 9.4). In these versions, both the `+` and `1+` are performed in *up* phases of the program. As

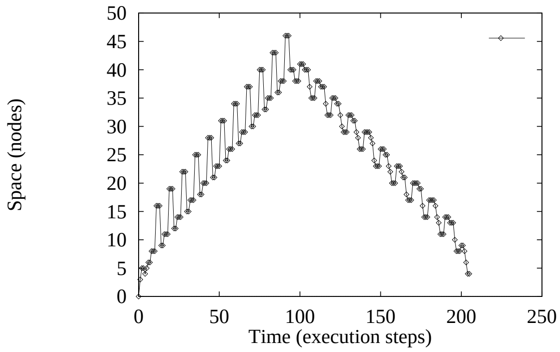
```
(define (average-file-up:mono filename)
  (let ((port (open-input-file filename)))
    (define (recur)
      (let ((elt (read port)))
        (if (eof-object? elt)
            (begin (close-input-port port)
                   (cons 0 0))
            (let ((sum&count (recur)))
              (cons (+ elt (car sum&count))
                    (1+ (cdr sum&count)))))))
    (let ((sum&count (recur)))
      (/ (car sum&count)
         (cdr sum&count)))))

(define (average-file-up filename)
  (let ((nums (splay-file filename)))
    (/ (upQ 0 + nums)
       (upQ 0 inc nums))))

;;; INC and SPLAY-FILE the same as before.
```

Figure 9.4: Monolithic and modular versions of a recursive program for averaging the contents of a numeric file.

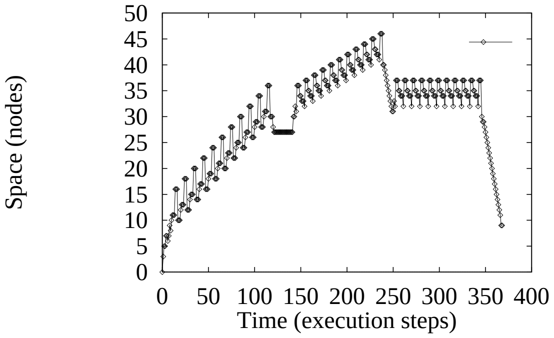
expected, all versions require space linear in the size of the file due to the stacked operations and their arguments (details omitted).

However, it is still possible to classify the *up* programs by the qualitative nature of their individual space profiles (Figure 9.5). As in the *down* case, there are three categories. In the monolithic, concurrent lazy, and prioritized SYNAPSE profiles, a rising (stack-pushing)

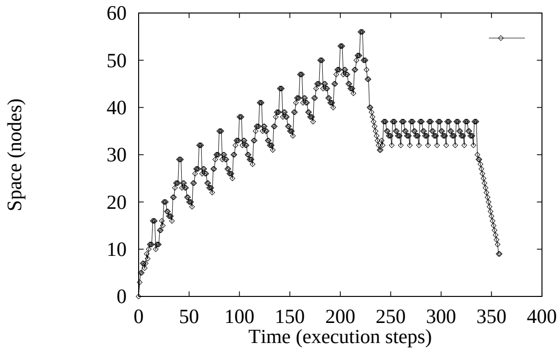
[8] More precisely, the rise is due to the creation of a stack of pending `cons` operations. This is followed by a brief stack-popping phase in which the pending `cons` operations are executed to create a list. This stack-popping phase appears as a short straight line segment in the profile at the beginning of the plateau.



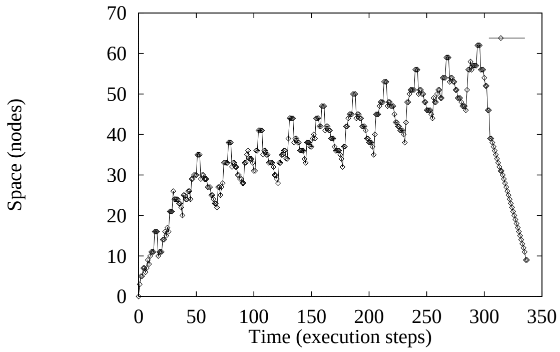
(a) Concurrent monolothic



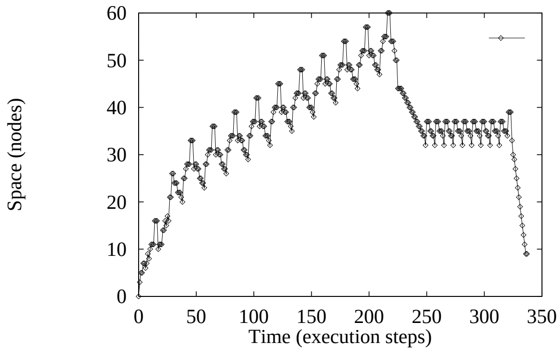
(b) Sequential strict



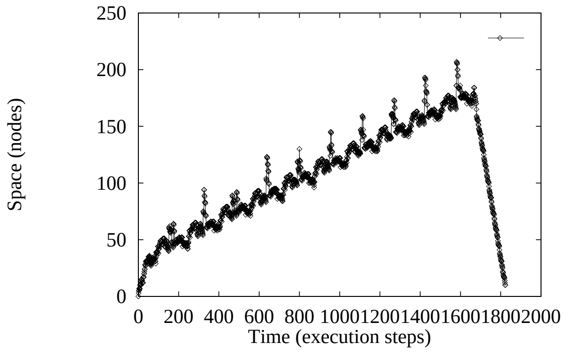
(c) Sequential lazy



(d) Concurrent lazy



(e) Prioritized concurrent lazy



(f) Prioritized SYNAPSE

Figure 9.5: Space profiles of various *up* averaging programs on an input file of ten elements.

phase is immediately followed by a falling (stack-popping) phase. In the sequential lazy and prioritized concurrent lazy versions, the initial rise (combining list-creation and the first stack-pushing) is followed by a sharp decline (the first stack-popping accumulation), a plateau (turning a list data structure into a stack for the second accumulation), and another sharp decline (the second stack-popping accumulation). These same three falling stages are found in the sequential strict case, but its rising phase is split into two independent parts: list-creation and the first stack-pushing.

The bottom line of the averaging experiments is that the modular SYNAPSE programs accurately model the qualitative space behavior of the monolithic averaging programs both at the coarse-grained (order of growth) and fine-grained (space profile) levels. While a concurrent lazy program is *likely* to show the same desirable qualitative space behavior, it is not *guaranteed* to (as shown by the prioritized example).

Alpha Rename

To test the space behavior of SYNAPSE for a non-trivial tree example, an order of growth experiment was performed on a modified alpha renamer. In order to accurately measure the space required by the alpha renaming process, it is necessary to subtract off the space taken up by the input and output terms. To remove the space associated with the output term, the monolithic alpha renamer in Section 2.2 and the modular version in Section 6.2.3 were modified to return a *parallel up* accumulated sum of the number of variable references in the alpha renamed term. In order to discount the size of the input terms, they were chosen to be ones that could be represented in space linear (rather than exponential) in their depth. (Thus, the input terms were not eliminated, but made sufficiently small.) In particular, they were chosen to be of the form

$$term_i = \texttt{(lambda x } body_i\texttt{)}$$

where $body_i$ is recursively defined by

$$body_0 = \texttt{x}$$

$$body_n = \texttt{(call } body_{n-1}\ body_{n-1}\texttt{)}$$

Using sharing of subterms, it is easy to generate a version of $body_i$ that uses space linear in i.

Figure 9.6 shows a log-log plot for the space required by various versions of the modified alpha renamer. The x-axis measures the depth of the input tree $term_i$. The y-axis measures the maximal number of nodes required by the computation. Simple analysis shows that the strict and lazy approaches require space exponential in the depth of the input tree, while the monolithic and SYNAPSE approaches require space linear in the depth of the tree.

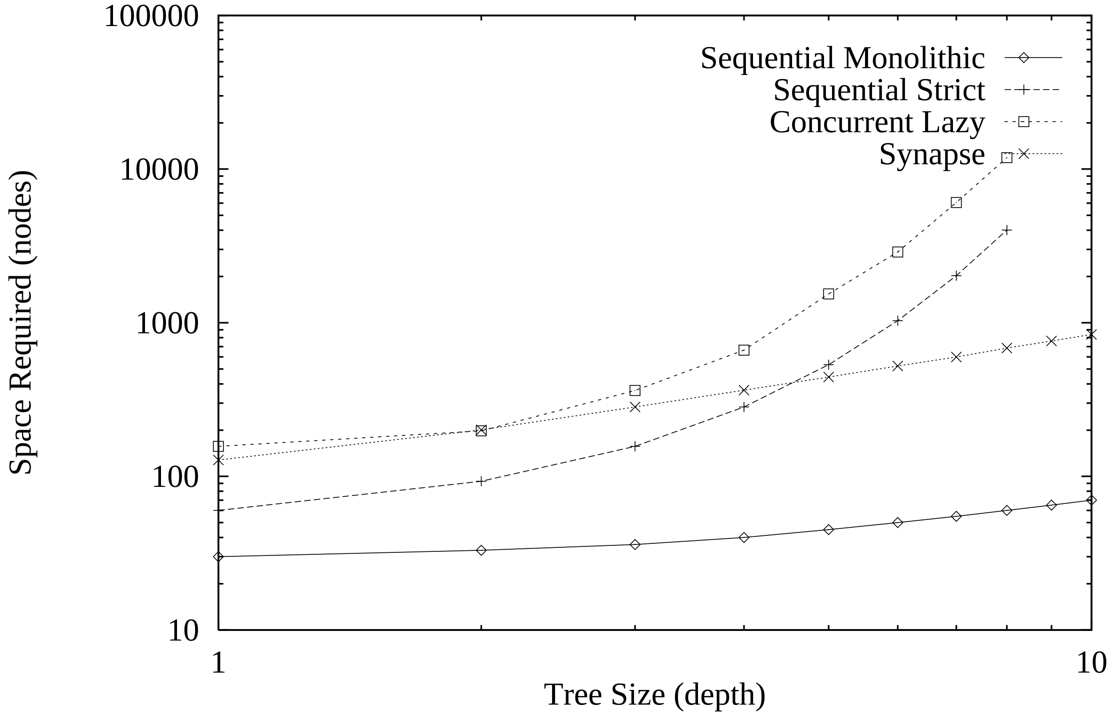


Figure 9.6: Log-log plot of the space consumed by various versions of the modified alpha renaming program as a function of the depth of the input tree.

Once again, the evidence is that the SYNAPSE program accurately models the desirable qualitative behavior of a monolithc program. Granted, in the current prototype system, the SYNAPSE version of the modified alpha renamer requires an order of magnitude more time and an order of magnitude more space than the monolithic program. Nevertheless, the SYNAPSE program is able to handle inputs far larger than the strict or lazy approaches, which both exhaust the OPERA interpreter's memory for input depths 9 or greater. Moreover, it is arguably easier to write the SYNAPSE program than the more complex monolithic program. SYNAPSE programs may be slow, but they are infinitely faster than programs that were never written because they were too hard to understand!

9.3 Lessons

Here I summarize several important lessons that I have learned while designing and implementing slivers. While these lessons are not necessarily new, they are especially highlighted by this research.

1. *Concurrency is a crucial modularity mechanism*: Concurrency is often touted as a means of making programs more efficient. While efficiency may be of practical interest, this perspective clouds the *real* advantage of concurrency: modularity. Concurrency makes it possible to express complex process behaviors as the combination of simpler process behaviors. Without concurrency, complex process behaviors can be achieved only by laboriously interweaving separate process specifications by hand.

   Concurrency has proven to be a cornerstone of the sliver technique. For several years I unsuccessfully attempted to develop sliver-like modularity mechanisms in sequential models. Eventually, it became apparent that concurrency was necessary for controlling space behavior operation order in a modular way. I was later pleased to find that Hughes [Hug84] had come to a similar conclusion a decade ago!

   In retrospect, the notion that concurrency is important to modularity has been around for a long time; it's just not emphasized enough. The notion of concurrently executing modular units has cropped up repeatedly in compiler optimization, operating systems, robotics, and data analysis. Perhaps unfortunately, limited versions of concurrency can be expressed within fundamentally sequential programming models by techniques such as laziness, coroutining, and continuations. While these techniques are important, I think they tend to support the illusion that full-fledged concurrency is not important. Perhaps the emerging emphasis on multi-threading in a number of current languages will help to start moving programmers out of the sequential mindset.

2. *Visualization tools are essential for concurrent programming*:

   According to folklore, concurrency and synchronization are hard to think about. My experience shows that the folkore is absolutely correct. After I latched onto the concurrent paradigm and developed the notion of synchrons, I quickly stumbled in my early efforts at implementating slivers. Deadlock was an ever-present problem. Discovering the cause of a deadlock in a large graph using simple text-based tools could easily take on the order of hours.

   Early debugging nightmares convinced me to bite the bullet and create a program animator that visually displayed program graphs. This was probably the best decision made during the entire project. The DYNAMATOR took a few months to build, but quickly paid itself off in saved debugging time. Bugs that used to take hours to find could now be discovered in minutes. More important, the DYNAMATOR helped forge new intuitions that fed back into the design process. In retrospect, I do not see how I could have ever converged on the final EDGAR rules or managed all the subtle complexities of the SYNAPSE implementation without the aid of the DYNAMATOR.

   Interestingly, the DYNAMATOR has an important drawback. I have come to rely so much upon it that I can't imagine adding new features to EDGAR without also adding them to the DYNAMATOR. Unfortunately, this means that those additions requiring non-trivial extensions to the DYNAMATOR are less likely to be implemented. In particular, the DYNAMATOR currently assumes that every node has a single output port. Yet, some important features, like continuations and multiple-value returns, require nodes with multiple output ports. The implementation of these features has been delayed because they require major extensions to the DYNAMATOR.

3. *Laziness requires defensive programming strategies*:

   Although I expected concurrent programming to be hard, I was not prepared for the wrench that non-strictness (especially laziness) would throw into the works. The principle of laziness is simple enough – things shouldn't be evaluated until (and unless) they are actually needed. Laziness is powerful because it allows the creation of conceptually infinite data structures and otherwise unresolvable circular dependencies. However, even without the complications of concurrency and synchronization, laziness can be very difficult to reason about.

Consider the following two lambda expressions written in OPERA:

(a)
```
(lambda (x)
  (lazon (if (and (= (length x) 2)
                  (eq? (car x) 'bar))
             (list 'bar 'quux)
             (list 'bar 'baz))))
```
(b)
```
(lambda (x)
  (list 'bar
        (lazon (if (and (= (length x) 2)
                        (eq? (car x) 'bar))
                   'quux
                   'baz))))
```

Both return a list of two elements whose first element is always the symbol `bar`. The first returns this list lazily; the second factors out the common `bar` and returns only the second element of the list lazily. While at first glance it might appear that the differences between these two expressions are unimportant, they can make a crucial difference in certain contexts, like the following:

```
(cadr (letrec ((foo E)
               (lst (lazon (foo lst))))
        lst))
```

If the first lambda expression is substituted for E, the resulting expression deadlocks. If the second lambda expression is substituted for E, the resulting expression terminates normally with the value `quux`.

The deadlock in the first case is not caused by concurrency or synchronization, but by an unresolvable circular dependency. To return a result, the `if` is first required to find the `length` of `x` and the `car` of `x`; but `x` turns out to be the output of the `if` expression! The unresolvable dependency is avoided in the second case by aggressively constructing the parts of the output that the `if` must examine (i.e., a two-element list whose first element is `bar`). While the above example is contrived, I have encountered many similar situations in my implementation of SYNAPSE.

Aggressive construction of data structures in the presence of laziness is an example of what I call *defensive programming strategies*. A defensive strategy aims to express known invariants to the interpreter as soon as possible in a program. In the above example, the invariant is that the result of the lambda expression is a two element list whose first element is `bar`. Another example of a defensive programming strategy is the aggressive unbundling requirement discussed in Section 5.4.2. In aggressive unbundling, the goal is to express which components of a data structure will later be used. These sorts of defensive programming strategies have proven invaluable in the implementation of SYNAPSE, where concurrency, synchronization, and eagerness compound the kinds of subtle deadlock problems that can be exhibited with laziness alone.

4. *Environment models are problematic*:

   This research has underscored some of the problems with traditional environment model for representing closures. A closure is a representation for a first-class procedure that pairs the code of the procedure with some structure that determines the meaning of free variable references that appear in the code. In the classical environment model, such as that expounded in [ASS85], the meaning of free variables is determined by an environment structure that contains bindings for the names of all lexically enclosing variables. The problem with this approach is that the interpreter can inadvertently hold onto unreferenceable structures via the environment. In Scheme, this can result in insidious space leaks (especially in the case of stream programs). In OPERA, this problem would be particularly acute, because deadlock is virtually guaranteed if the interpreter does not release synchron-holding structures as soon as possible.

   Avoiding the problems of spuriously held structures requires new language models and implementation techniques. OPERA attacks this problem by using free variable analysis and lambda lifting to reduce the number of variables that a procedure closes over, and by using reference counting garbage collection to aggressively reclaim unreferenceable objects. These techniques aren't always viable. With the development of more sophisticated language constructs that are tied into the storage management model (like synchrons), other techniques for aggressively dropping references to objects will have to be developed.