Synchronized Lazy Aggregates

Franklyn Turbak
lyn@lcs.mit.edu

Laboratory for Computer Science,
Massachusetts Institute of Technology

Draft of December 1, 1993

Abstract

We present a new solution to the problem of expressing single-pass, space-efficient list and tree computations in a modular fashion. *Synchronized lazy aggregates* augment existing methods for expressing programs as networks of operators acting on aggregate data, but overcome many limitations. The technique is based on a dynamic model of lock step processing that enables the operators in a network to simulate the operational behavior of a monolithic loop or recursion. The key to the technique is the *synchron*, a new first-class object that allows concurrently executing operators to participate in a barrier synchronization. We describe the design of synchronized lazy aggregates and present examples of their implementation and use.

1 Introduction

Ideally, programming languages should encourage programmers to express their designs in a modular fashion based on libraries of mix-and-match components. But classic modularity mechanisms are typically at odds with the desire of programmers to control important operational aspects of their programs. These mechanisms help programmers build programs that have the desired *functional* behavior, but not necessarily the desired *operational* behavior. As a result, programmers often eschew modularity in order to control the operational details of their programs.

Consider the problem of expressing a single-pass, space-efficient tree algorithm as the modular superposition of simpler tree walks. Such an algorithm can often be viewed as a signal processing system in which a network of devices that represent the tree walks communicate via wires that transmit tree-shaped data. We shall call any programming technique that reflects this modular organization a *signal processing style (SPS)* technique. Unfortunately, existing SPS techniques often fail to preserve desirable time and space properties of the original algorithm. Whereas a monolithic (i.e., non-modular) tree walk typically requires space proportional to the depth of the tree, it is common for a modular version to either walk the given tree multiple times or store intermediate trees as large as the given tree. Practically, the extra time or space overhead of the modular version may be unacceptable. But more fundamentally, traditional modularity techniques unduly restrict the range of computations that programmers want to express.

Synchronized lazy aggregates are a mechanism that helps to relax the tension between modularity and control. They are based on the following idea. Suppose that several list or tree traversal idioms are interwoven in a monolithic recursive procedure. Then it should be possible to distribute the recursive call structure of the monolithic procedure across the idioms to yield a network of communicating modular subprocedures (see Figure 1). In the monolithic recursion, each procedure call[1] controls behavior by coordinating the different idioms. But this coordination is sacrificed when the monolithic call is split into the many calls of the subprocedures. Synchronized lazy aggregates reestablish coordination by requiring the calls of communicating subprocedures to "line up" (see Figure 2). The alignment is achieved by extending aggregate

[1] We assume that all procedure calls are *strict* — i.e., all argument expressions must be evaluated to values before the procedure body is evaluated. Strictness simplifies reasoning about space consumption and the relative order of operations.

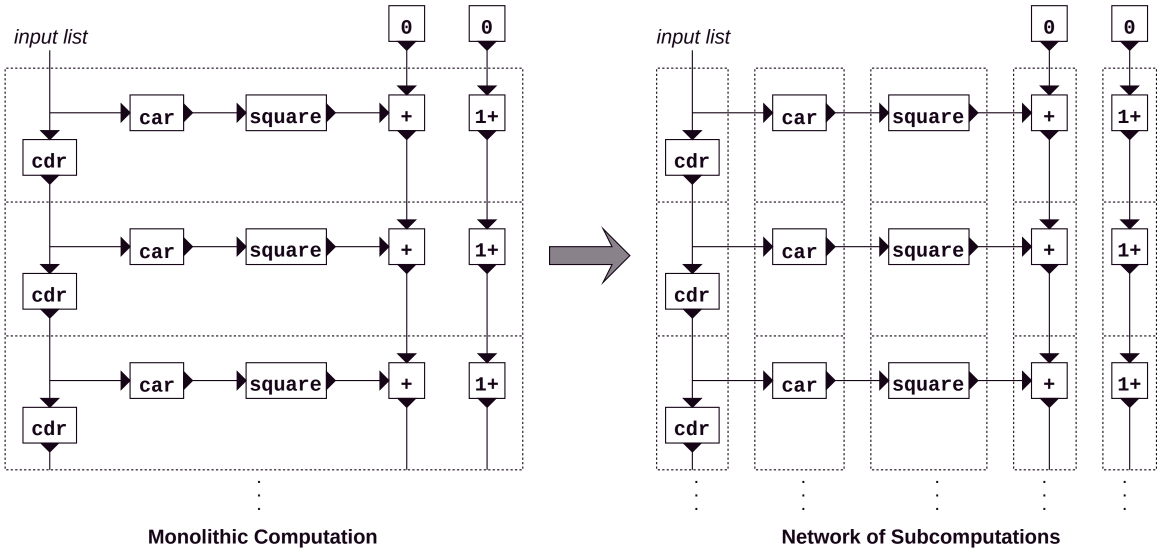

Figure 1: Decomposing a monolithic computation into communicating subcomputations. The labelled boxes represent primitive operations performed while iteratively computing the average of the squares of an input list (the conditional handling of empty list tests is not depicted). The dotted lines represent procedure call boundaries of the monolithic computation that are distributed over the subcomputations.

data structures to carry synchronization objects called *synchrons*, and requiring connected subprocedures to rendezvous at the synchrons at every corresponding subprocedure call and return.

Synchronized lazy aggregates are more expressive than existing SPS techniques because they permit fine-grained operational control in a framework that supports tree-structured data, arbitrary network topologies, imperative features, and hierarchical program organization. Synchronized lazy aggregates resemble Waters's series package for Common Lisp [Wat90, Wat91], except that (1) they can express general linear and tree recursions in addition to iterations and (2) they provide a dynamic model of lock step processing rather than a static one.

The remainder of this paper is organized as follows: Section 2 lists the goals for this work and explains why existing techniques fail to achieve these goals; Section 3 gives a principled design of synchronized lazy aggregates; Section 4 gives examples of the technique; Section 5 discusses our experience with the technique; and Section 6 concludes with a summary and an outline of future work.

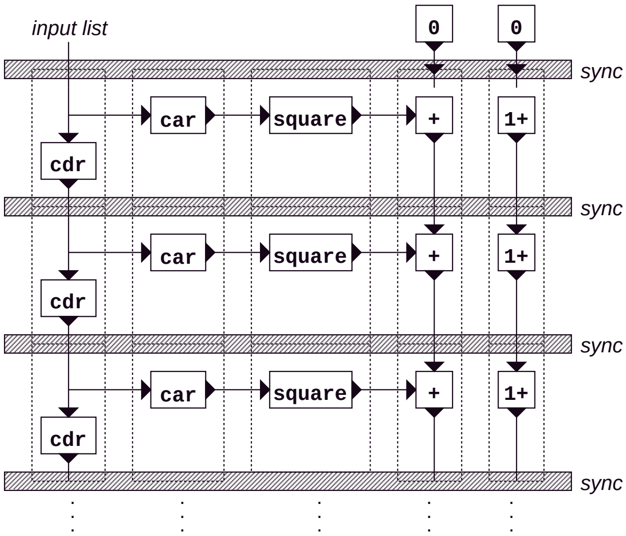

Figure 2: Synchronization events (shaded barriers labelled *sync*) can force the alignment of all corresponding procedure call boundaries within a network.

2 Goals

Addressing operational concerns within the signal processing style requires a technique that meets the following goals:

- *Operational control*: The technique should enable the control of essential time and space characteristics of an SPS network in a modular fashion. In particular, for any network that obviously corresponds to a monolithic recursion, it should be possible to guarantee that the network exhibits the same *time profile* and the same *space profile* as the monolithic recursion. A time profile indicates the relative order of the basic operations performed by a computation. A space profile indicates the working space required by a program as its computation unfolds. The technique should be allowed to employ additional operations and storage for management purposes as long as it maintains the desired order of growth of a computation in space and time.

- *Tree-structured communication*: The technique should be powerful enough to express the tree-shaped nature of general recursions in a modular fashion. Modularity implies the need to communicate intermediate values produced by one component of a recursion to another component of the recursion. Since the intermediate values of a general recursion are naturally arranged in trees,[2] the technique should allow wires to transmit tree-structured data. Lists are an important subclass of trees that may require special handling (e.g., to model loops).

- *General network topologies*: Many SPS programs naturally exhibit *fan-in* (where a device consumes multiple inputs), *fan-out* (where a device produces multiple outputs or the same output is consumed in multiple places), and *cycles* (where there is a path from an output of a device back to one of its inputs). The technique should support these kinds of connections.

- *Reusability*: Devices should share a standard interface so that they can be recombined in a mix-and-match way to model a wide range of computations.

- *Hierarchical, nested structure*: To model the hierarchial structure of many programs, the technique should permit SPS devices themselves to be represented as networks of subdevices. In addition, it should be possible to apply the technique recursively to individual elements of the structures transmitted by wires (e.g., to handle lists of lists, trees of trees, etc.).

Existing SPS techniques fail to meet one or more of these goals. In the *aggregate data approach*, SPS devices are represented as operators (e.g., generate, map, filter, accumulate) that manipulate aggregate data structures (e.g., lists, trees, arrays). This approach includes APL's array operators [Ive87], Common Lisp's series procedures [Wat90], Scheme's stream procedures [ASS85], and Haskell's list comprehensions [HJW$^+$92]. The major problem with existing aggregate data techniques is their inability to express operational control. Strict operators processing large intermediate structures in a staged fashion cannot simulate the time or space profiles of a monolithic recursion. The operator interleaving provided by laziness [Hug90] suitably handles tree-structured networks, but interacts poorly with fan-out. In fact, Hughes shows that any *sequential* functional language must exhibit undesirable buffering for networks with fan-out [Hug83]. Transformations that remove intermediate aggregates from a program either provide no guarantees as to what will be removed (e.g., [Bel84]) or limit the class of expressible computations (e.g., Waters's series compiler [Wat90] handles only iterative computations; deforestation techniques [Wad88] disallow fan-out).

In the *channel approach*, devices are wired together by some sort of explicit communication channel. This approach encompasses communicating threads (e.g. [Bir89], CSP [Hoa85], ML threads [CM90]), producer/consumer coroutines (e.g., Unix pipes [KP84], CLU iterators [L$^+$79], Id's I-structures [ANP89] and M-structures [Bar92]), and dataflow techniques (e.g., [Den75], [WA85]). Many channel techniques disallow fan-out, recursion, nesting, or the transmission of tree-structured

[2] We assume the absence of control structures like non-local exits and continuations, which can lead to more general graph structures.

data.[3] The space profiles of monolithic recursions can often be modelled by bounded channels, but the device buffering and loose coupling exhibited by many channel networks makes it difficult to simulate their time profiles.

Modular attribute grammars [DC90, FMY92, Ada91] specify tree decoration programs in a declarative, modular fashion. Modular attribute grammars allow different forms of tree processing to be interwoven. Although some versions limit number of traversals or space consumption, most approaches provide little control over these fundamental operational issues. Also, attribute grammar formalisms are often closely tied to parsing technology, which limits their use for general SPS programs (but see [Joh87]).

3 Design

We describe the design of a new SPS technique that satisfies all of the goals set forth above. To guide the design, we add the constraint that the technique should support the traditional aggregate data style of programming. That is, we introduce a new class of aggregate data types and associated operators, but programmers can manipulate these with the same familiar programming methods used for aggregate operations on lists and trees. We shall see that achieving this goal requires a base language with appropriate support for laziness, concurrency, and synchronization.

Synchronized lazy aggregates (abbreviated *slags*) augment lazy aggregates [Hug90] with a mechanism for aligning the recursive call structures of networked operators to simulate the behavior of a monolithic recursion. In conjunction with laziness, the lock step nature of the operators facilitates the control of time and space profiles, while the aggregate aspect supports hierarchy, nesting, tree-shaped data, general connection topologies, and reusability. The lock step processing implied by slags is not appropriate for many computations. However, the tight coupling of slags can be flexibly mixed with the loose coupling afforded by other mechanisms (such as lazy lists and trees [ASS85, Hug90]).

Synchronized lazy aggregates are similar in spirit to Waters's series extension to Common Lisp [Wat90, Wat91]. However, whereas series can express only iterations, slags can express general linear and tree recursions. There is also a difference in focus between the systems. Emphasizing the efficient compilation of series expressions into loops, Waters develops a *static* model for the lock step processing of series operators. In contrast, issues of expressiveness motivate us to develop a *dynamic* model for the lock step processing of slag operators.

The mixture of concurrency, synchronization, and laziness on which slags are based resembles Hughes's technique for controlling space consumption in modular functional programs [Hug83, Hug84]. However, the channel-based nature of Hughes's technique (1) makes it difficult for an SPS network to simulate the time profiles of monolithic recursions and (2) requires programming in a style that is not as straightforward as the aggregate data approach.[4] Moreover, while Hughes is motivated by the desire to take advantage of parallelism in a functional language, we focus solely on how to modularly express the fine-grained behavior of monolithic recursions in a procedural language.

3.1 A Lock Step Processing Model

We begin with the assumption that an SPS network should be decomposable into loosely coupled assemblies of tightly coupled subnetworks. Within a tightly coupled subnetwork, all operators are intended to work in lock step to simulate the behavior of a monolithic recursion. The kind of data transmitted by a wire connecting two devices indicates the desired coupling between the devices: slags indicate a tight coupling, while classical strict and lazy data indicate a looser coupling.

Intuitively, each tightly coupled subnetwork is the connected component of the graph obtained by erasing all non-slag wires from a given network. For this reason, we use the term *slag component* to refer to such a subnetwork. Because the structure of a general SPS network may not be known until run

[3] There are methods of encoding trees as linear streams, but manipulations of the resulting streams often don't accurately reflect the tree-shaped nature of the corresponding monolithic computations.

[4] In particular, Hughes's technique requires programmers to use an explicit copy operator for every instance of fan-out in an SPS network.

time, slag components are a dynamic characteristic of a computation. We will assume that every slag component is a directed acyclic graph (DAG) of devices and slag wires. Cyclic SPS networks can be handled as long as every cyclic path contains at least one non-slag wire.

To model the time and space profiles of a monolithic recursion, all of the operators within a slag component should work together to simulate the computation generated by a recursive procedure. We assume that each slag operator is itself a recursive procedure and that the recursion patterns of all the operators are compatible (i.e., for every call in one procedure, there is an obvious corresponding call in all the others).

The lock step processing of slag operators involves both *communication* and *synchronization*. We assume a demand-driven communication in which an operator at the target of a wire can request an element from the device at the source of a wire. Synchronization is achieved by having all slag operators participate in a *rendezvous* at their corresponding calls and returns. The rendezvous serves as a *barrier synchronization* that prevents an individual operator from proceeding until all the operators are ready to proceed. Each rendezvous represents the call or return of a monolithic recursive procedure that interleaves the processing of all the operators of a slag component.

A slag operator that makes a *tail call* [Ste77] never returns from the call and so should not participate in the barrier synchronization associated with the return. If all operators within a slag component make corresponding tail calls, then there is no global return associated with the call. Thus, an entire slag component can exhibit the tail recurisive behavior of its operators. This feature is critical for simulating the space profiles of iterations and other computations expressible with tail calls. For example, a slag component consisting purely of iterative operators is guaranteed to exhibit the constant space requirements of a monolithic iteration.

3.2 Linguistic Requirements

The lock step processing model sketched above implies that the slag technique is only viable in a base language that provides certain laziness, concurrency, and synchronization features:

- *Laziness*: The demand driven nature of communication implies that the structure and elements of a slag are computed only when needed. A base language must allow the expression of delayed computations in order to represent the lazy nature of slags.

- *Concurrency*: The lock step processing of a slag component requires its operators to execute concurrently. As shown by Hughes [Hug83, Hug84], the coroutining behavior associated with demand driven evaluation strategies is an insufficient form of concurrency for obtaining lock step behavior in the presence of fan-out. A base language must provide a more general form of concurrency that allows fine-grained interleaving of operations among the operators of a slag component.

  The concurrency requirement is motivated purely by modularity concerns, *not* by any desire to take advantage of multiple physical processors. So multi-tasking on a single physical processor is sufficient for our purposes.

- *Synchronization*: The barrier synchronization among operators at calls and returns requires a novel form of synchronization.[5] In traditional barrier synchronization [Axe86], the number of participating processes is known in advance, and synchronization can easily be implemented by a counter. However, due to the dynamic nature of slag components, the number of operators participating in the barrier synchronization cannot generally be predicted from the program.

  To deal with this problem, we introduce a new first-class synchronization entity, the *synchron*, that ties barrier synchronization to automatic storage management. Pointers to a synchron are classified into two types: *waiting* and *non-waiting*. When a process wishes to rendezvous at a synchron, it enters a waiting

[5] Due to atomicity problems introduced by concurrency, a base language with imperative features also needs to support a traditional locking mechanism [Bir89], but we won't focus on this point.

state that refers to the synchron with a distinguished waiting pointer. For all other manipulations, a process holds a synchron with a non-waiting pointer. The synchronization condition is this: *a rendezvous occurs at a synchron only when all of the pointers to it are waiting pointers.* This means that any non-waiting pointer to a synchron effectively blocks a rendezvous. Since all processes lose access to the synchron after the rendezvous, there can only be one rendezvous per synchron.

The rendezvous protocol of synchrons sets it apart from other synchronization structures (e.g., semaphores/locks [Bir89], synchronous messages [Hoa85], I-structures [ANP89], and M-structures [Bar92]). Synchronization typically involves some processes *waiting* in a suspended state for a shared synchronization entity to be *released* by the process that currently owns it. Traditional protocols supply explicit *wait* and *release* operations. With synchrons, only the *wait* is explicit; the *release* is implicitly handled by the automatic storage manager.

Synchrons support three operations: *create* a synchron; *wait* for a rendezvous at a synchron; and *unify* two synchrons. Synchron unification extends the usual notion of unifying logic variables to unifying the points in time represented by independently generated synchrons. Below, we shall see how networks with fan-in require this feature. Synchrons can be formally specified by a graph rewriting model; see [Tur94] for details.

3.3 Slag Structure

The lock step processing of a slag component implies that all of its operators share the same synchron for a given call or return. This sharing is achieved by the generation, propagation, and combination of synchrons. A synchronized lazy aggregate is a lazy aggregate data structure that manages synchrons in addition to its component elements. Slag operators communicate via the elements of a slag and they synchronize via its synchrons.

In a classical aggregate, component *elements* are leaves hanging off of a framework of *structural nodes*. In a *synchronized* aggregate, every pointer to a structural node is decorated with a pair of synchrons that represent a call and a return for a particular procedure invocation; we call this pair a *barrion*. Figure 3 shows the relationship between a classical aggregate and the corresponding synchronized aggregate. The structural node depicted in the figure has only a single subnode, but the same idea generalizes to structural nodes with multiple subnodes.

A synchronized *lazy* aggregate is a synchronized aggregate in which the computation of elements and structural nodes is delayed until they are required. Laziness means that the slag dynamically unfolds over time. The dynamically computed elements of a slag correspond to the time-dependent values of a variable in a monolithic recursion. Waters uses the term *temporal data structure* [Wat91] to refer to this notion. His series data structure is a particular instance of the more general slag. Whereas a series represents the successive values of the state variable of a loop, a slag can represent the conceptual tree of values taken on by an identifier within an arbitrary recursive procedure. Another way to say this is that a series corresponds to a register while a slag correspond to a register plus a stack.

Slags can be also be viewed as a hybrid between lazy aggregates and synchronous communication channels for concurrent processes. Like lazy aggregates, slags are compound, potentially tree-shaped, data structures whose parts are not computed until they are required. Like synchronous communication channels, slags synchronize separate threads of control and manage inter-process storage resources.

3.4 Operator Requirements

To ensure that a slag component behaves like a monolithic recursion, all slag operators must obey the following requirements:

- *Synchron propagation*: To ensure that all operators in a slag component share the same synchron for a given call or return event, slag operators must appropriately generate, propagate, and combine the synchrons held by slags. A slag generator must create a fresh pair of synchrons before every call, and insert these

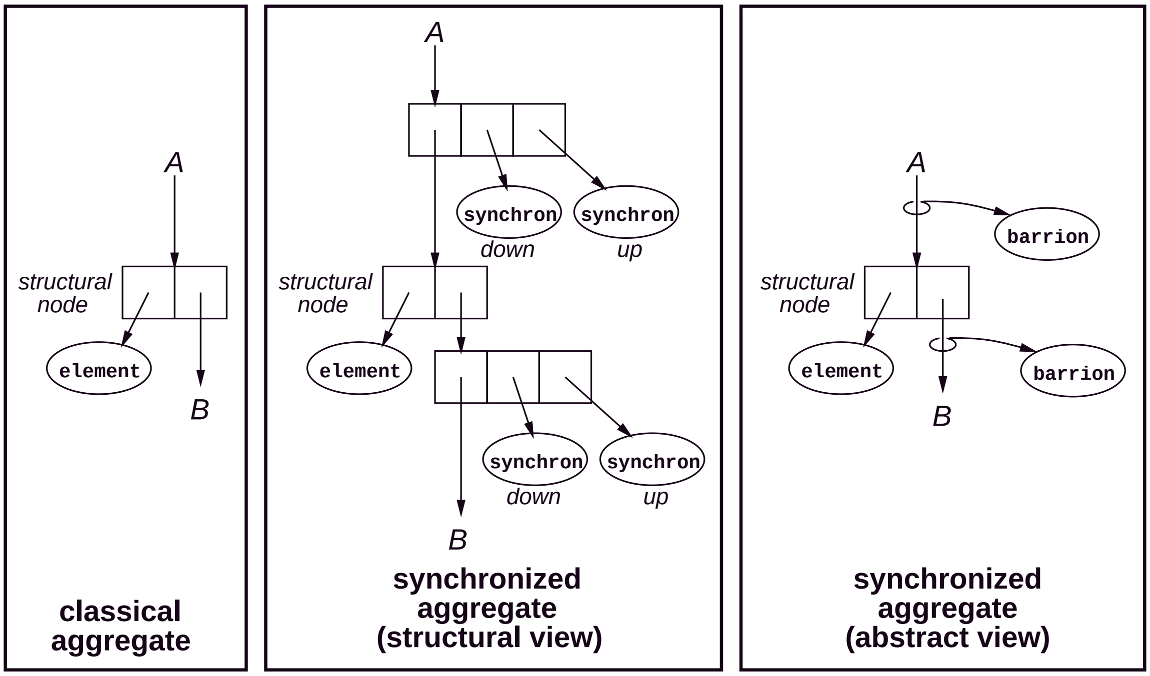

Figure 3: Synchronized aggregates are data structures decorated with synchrons. A *down* synchron represents a call, while an *up* synchron represents the corresponding return.

into all of its output slags. A slag transducer (i.e., operator mapping input slags to output slags) must preserve the identity and ordering of input slag synchrons in its output slags. If a transducer has multiple input slags (which may have been produced by independent generators) it unifies corresponding input synchrons into a single output synchron (via the synchron *unify* operation).

- *Down synchronization*: As the very last step before making a call, an operator must rendezvous (via the synchron *wait* operation) with all other operators sharing the current down synchron.

- *Up synchronization*: If an operator makes a non-tail call, then, as the very first step upon returning, it must rendezvous with all other operators sharing the current up synchron. An operator that makes a tail call never returns, so it will not wait on the up synchron.

- *Strict arguments/lazy elements*: To accurately model the time profiles of strict monolithic procedures, the procedures representing slag operators must evaluate all non-slag *arguments* in a strict fashion. Interestingly, faithful modelling also dictates that any slag-producing procedure should delay the computation of every slag *element* so that an element value is never computed unless it is requested.

- *Aggressive unbundling*: To preserve the storage behavior of a monolithic procedure and avoid spurious deadlocks, slag operators must aggressively unbundle slags into their component parts. For example, if an operator needs only one component of a slag, it must extract that component as soon as possible so that the other components become inaccessible (and can therefore be garbarge collected).

4 Examples

This section presents some simple examples of synchronized lazy lists (SLLs) and synchronized lazy trees (SLTs). Because their operational handling is intricately intertwined with their structure, slags resist being characterized as abstract data types; we have not been able to find a small set of operators that completely specifies the essence of slags. Instead, we have developed a suite of higher order slag operators that have proven useful in a wide variety of simple applications. Below, we show examples of some of these operators. Due to space limitations, the examples are necessarily brief; for more extended examples, see [Tur94].

All of the examples described here are written in OPERA, a dialect of Scheme that supports the laziness, concurrency, and synchronization features outlined in Section 3.2. For present purposes, the only interesting detail of OPERA is that the default evaluation strategy evaluates all subexpressions of a procedure call expression in parallel.[6]

4.1 Synchronized Lazy Lists

Figure 4 presents a sampler of higher order procedures on synchronized lazy lists. We defer a discussion of the implementation of these procedures until Section 4.3.

We will test an iterative and recursive SLL accumulator via the following `test` procedure:

```
(define (test accumulator list)
  (accumulator
   (mapL square
         (filterL positive?
                  (splay-list list)))))

(define (splay-list list)
  (mapL car (genL list cdr null?)))
```

`Test` applies the given accumulator to the SLL consisting of the squares of the positive numbers in the given list. The `splay-list` utility converts a traditional list into a SLL.

An iterative averaging operator can be defined as follows:

```
(define (down-average sll)
  (/ (down-+ sll) (down-length sll)))

(define (down-+ sll)
  (downL 0 + sll))

(define (down-length sll)
  (downL 0 (lambda (ignore len) (1+ len)) sll))
```

[6] This evaluation strategy is not allowed in standard Scheme [CR+91], which requires that the subexpressions be evaluated in some sequential order.

(`genL` *init next done?*) Generates a SLL whose first element is *init* and each of whose subsequent elements is determined from the previous by *next*. The SLL terminates when *done?* is true of the current element (which is not included in the SLL).

(`mapL` *fun sll*) Returns the SLL resulting from the elementwise application of *fun* to *sll*.

(`filterL` *pred sll*) Returns the SLL in which every element satisfying *pred* is mapped to itself and all other elements are mapped to *gaps* (objects that represent "holes" in a SLL).

(`map2L` *fun sll1 sll2*) Returns the SLL resulting from the elementwise application of *fun* to corresponding elements of *sll1* and *sll2*. An output element is a gap if either input element is a gap. The length of the output SLL is the shorter of the length of the two input SLLs.

(`truncateL` *pred sll*) Returns the prefix of *sll* up to, but not including, the first element for which *pred* is true.

(`nthL` *index sll*) Returns the result of iteratively calculating *index*th element (0-based) of an sll. An error is signalled if *index* is greater than the length of *sll*.

(`downL` *init combine sll*) Returns the result of iteratively accumulating the elements of *sll* using the combiner *combine* and the initial value *init*.

(`upL` *init combine sll*) Returns the result of recursively accumulating the elements of *sll* using the combiner *combine* and the initial value *init*.

(`down-scanL` *init combine sll*) Returns a SLL of intermediate accumulated values in the iterative accumulation of *sll* using combiner *combine* and initial value *init*.

(`up-scanL` *init combine synq*) Returns a SLL of intermediate accumulated values in the recursive accumulation of *sll* using combiner *combine* and initial value *init*.

Figure 4: Sample higher order operators on synchronized lazy lists. The "L" at the end of each name indicates a list (as opposed to tree) operator.

Note that `down-average` exhibits fan-out because its `sll` argument is used twice.

Here is a trace of a computation using `down-average`:[7]

```
OPERA> (test down-average '(2 -3 4))
-----------------------:down[A,0]
(null? (2 -3 4)) --> ()
(car (2 -3 4)) --> 2
(positive? 2) --> #t
(cdr (2 -3 4)) --> (-3 4)
(1+ 0) --> 1
(square 2) --> 4
(+ 4 0) --> 4
-----------------------:down[A,1]
(null? (-3 4)) --> ()
(car (-3 4)) --> -3
(positive? -3) --> ()
(cdr (-3 4)) --> (4)
-----------------------:down[A,2]
(null? (4)) --> ()
(car (4)) --> 4
(positive? 4) --> #t
(cdr (4)) --> ()
(square 4) --> 16
(+ 16 4) --> 20
(1+ 1) --> 2
-----------------------:down[A,3]
(null? ()) --> #t
(/ 20 2) --> 10
; Value: 10
```

The trace begins with an OPERA expression and ends with its value. The intervening lines show the OPERA primitives performed during the computation.[8] Each dotted line labelled `down` represents a rendezvous at a down synchron; the bracketed information identifies the synchron. The dotted lines partition the trace into episodes consisting of the interleaved operations from the corresponding recursive calls of each of the operators participating in the computation. Except for the presence of the dotted lines, the trace is indistinguishable from that of the obvious monolithic loop for performing the same averaging computation. The key feature of the trace is the interleaving of the `1+` and `+` operations; in a modular program based on Scheme streams (a kind of lazy list) [ASS85] rather than SLLs, fan-out would prevent these operations from being interleaved and would result in unnecessary buffering.

[7]The traces presented here were automatically generated by our prototype OPERA interpreter.

[8]They do *not* include the operations needed to manage slags.

A recursive (non-iterative) version of the averaging operator and its associated trace appear below:

```
(define (up-average sll)
  (/ (up-+ sll) (up-length sll)))

(define (up-+ sll)
  (upL 0 + sll))

(define (up-length sll)
  (upL 0 (lambda (ignore len) (1+ len)) sll))

OPERA> (test up-average '(2 -3 4))
-----------------------:down[A,0]
(null? (2 -3 4)) --> ()
(car (2 -3 4)) --> 2
(positive? 2) --> #t
(square 2) --> 4
(cdr (2 -3 4)) --> (-3 4)
-----------------------:down[A,1]
(null? (-3 4)) --> ()
(car (-3 4)) --> -3
(positive? -3) --> ()
(cdr (-3 4)) --> (4)
-----------------------:down[A,2]
(null? (4)) --> ()
(car (4)) --> 4
(positive? 4) --> #t
(square 4) --> 16
(cdr (4)) --> ()
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(+ 16 0) --> 16
(1+ 0) --> 1
-----------------------:up[A,1]
(+ 4 16) --> 20
(1+ 1) --> 2
-----------------------:up[A,0]
(/ 20 2) --> 10
; Value: 10
```

The most interesting feature of this trace is the presence of `up`-labelled dotted lines. These represent a rendevous among the operators that return from their local recursive calls. The trace for `down-average` does not have any such lines because all of its operators are iterative. But `up-average` inherits a stack-pushing behavior from its `up-+` and `up-length` components. Informally, each up line is "pushed" at the time represented by the down line with the same annotation, and they are popped in the opposite order. There is no line labelled `up[A,2]` because the corresponding call (in the case of a filtered element) is a non-returning tail call. The network as a whole exhibits the detailed operational structure of a recursion that returns both the sum and the length of the current list argument.

Though extremely simple, the above examples illustrate how modular slag programs can faithfully model the time and space profiles of monolithic recursions.

4.2 Synchronized Lazy Trees

Synchronized lazy trees support a suite of higher order operators that generalize the corresponding list operators. SLT generators, mappers, and filters are straightforward, but SLT accumulators and scanners support a much richer variety of processing shapes than their SLL cousins. For example, processing can proceed in parallel either down or up the branches of a tree, or it can proceed sequentially (e.g., left-to-right, right-to-left) in various traversal orders (pre-order, in-order, post-order). There are higher order SLT operators corresponding to each of these possibilities.

Space does not permit an enumeration of SLT operators or an examination of the traces that they generate. However, we can illustrate the power of SLT with a non-trivial example: alpha renaming. An alpha renaming program gives unique names to the logically distinct variables of a lambda calculus term. Here is an OPERA program that performs alpha renaming on a lambda calculus term represented as an SLT:[9]

```
(define (alpha-rename slt)
  (let ((fmls (mapT formal (filterT lambda? slt))))
    (map2T rename
           slt
           (down-scan2T env-empty
                        env-bind
                        fmls
                        (lr-pre-scanT 0
                                      count
                                      fmls)))))

(define (count ignore len) (1+ n))

(define (rename node env)
  (cond
   ((lambda? node)
    (make-lambda (env-lookup (formal node) env)))
   ((var? node)
    (make-var (env-lookup (name node) env)))
   (else node)))
```

[9]We have omitted the definitions of abstract syntax procedures (`lambda?`, `make-lambda`, `formal`, etc.) and environment procedures (`env-empty`, `env-bind`, `env-lookup`).

This alpha renamer uniquely numbers the formal parameters in a left-to-right pre-order tree walk, inserts parameter/number associations in a downward-extended environment structure, and consistently renames formals and variable references according to the corresponding environment.[10] `MapT`, `Map2T`, and `FilterT` are SLT mapping and filtering operators. `Down-scan2T` returns a SLT of partial values resulting from performing an accumulation down the corresponding branches of two SLTs; in this case it makes a tree of environments (structures that map names to values). `Lr-pre-scanT` returns a SLT of partial values resulting from performing a left-to-right pre-order accumulation on a given SLT.

Writing the above program in a monolithic fashion is challenging and leads to a program that is hard to read, modify, and reuse. The remarkable feature of the above program is that synchronized lazy trees guarantee that it exhibits the desirable space profile of the monolithic version even though it is written in a modular style. This lets the programmer concentrate on the high level structure of the program, encouraging reuse and experimentation. For example, the renaming strategy of the alpha renamer can be changed by modifying a single component: a deBruijn numbering program [Pey87] can be obtained from `alpha-rename` simply by replacing `lr-pre-scanT` with `down-scanT`.

4.3 Implementing Slag Operators

To show how slag operators can satisfy the requirements listed in Section 3.4, we present OPERA implementations of a few synchronized lazy list operators.

Slags are represented as lists that are assembled and disassembled via `bundle` and `unbundle`:

```
(define (bundle down up $structural-node)
  (list down up $structural-node))

(define (unbundle slag receiver)
  (receiver (car slag) (cadr slag) (caddr slag)))
```

The `$` in `$structural-node` is a naming convention indicating that the variable is expected to name a delayed computation. Here, it indicates that the structural node is held lazily by the slag.

[10]For simplicity, we assume that numbers are valid names.

The following abstractions are helpful for satisfying the down and up synchronization requirements:

```
(define (call/down down proc . args)
  (begin (wait down)
         (apply proc args)))

(define (call/down-up down up proc . args)
  (begin (wait down)
         (let ((result (apply proc args)))
           (wait up)
           result)))

(define (call/down-slag proc arg1 . rest)
  (unbundle arg1
    (lambda (down up $snode)
      (apply call/down down
             proc
             (future (touch $snode)) rest))))

(define (call/down-up-slag proc arg1 . rest)
  (unbundle arg1
    (lambda (down up $snode)
      (apply call/down-up down up
             proc
             (future (touch $snode)) rest))))
```

`Call/down` uses the synchron `wait` procedure to rendezvous with other operators before a tail call of `proc`. `Call/down-up` is similar, but also engages in a rendezvous upon return from `proc` to model a non-tail call. `Call/down-slag` and `call/down-up-slag` specially handle the case where the first argument is a slag. To allow demand to propagate through a slag component, it is necessary to eagerly evaluate the (otherwise delayed) structural node of the slag. Eager evaluation is achieved via *futures* (see [Hal85, Mil87]).

Figure 5 presents implementations of the SLL `genL` generator, `mapL` mapper, and `upL` accumulator. `GenL` bundles fresh synchrons (created by the `synchron` constructor) with a delayed computation that generates the next element of the SLL. A delayed computation is created by `lazon`, which is similar to Scheme's `delay` [ASS85] except that it is implicitly forced in contexts requiring its value. `MapL` uses the `rebundle` abstraction to propagate synchrons from its input SLL to its output SLL. The `gap?` predicate and gap constant `#g` are used to handle filtering; unlike lists, filtered SLLs are not compacted, so it is necesary to leave behind some sort of hole when an element is filtered out. `UpL` uses `call/down-up-slag` to perform a non-tail call; however, since a gap requires no pending accumulation, `upL` can use the non-returning

```
(define (genL init next done?)
  (define (gen-sll $arg)
    (let ((down (synchron))
          (up (synchron)))
      (bundle down up
        (lazon
         (call/down down gen-list (touch $arg))))))
  (define (gen-list arg)
    (if (done? arg)
        '()
        (cons arg (gen-sll (lazon (next arg))))))
  (gen-sll init))

(define (mapL fun sll)
  (rebundle sll
    (lambda (lst)
      (if (null? lst)
          '()
          (let (($elt (car lst))
                (rest (cdr lst)))
            (cons (if (gap? $elt) #g (fun $elt))
                  (mapL fun rest)))))))

(define (rebundle slag receiver)
  (unbundle slag
    (lambda (down up $snode)
      (bundle down up
        (lazon
         (let ((contents (future (touch $snode))))
           (call/down down
             (lambda () (receiver contents)))))))))

(define (upL init combiner sll)
  (define (accum lst)
    (if (null? lst)
        init
        (let (($elt (car lst))
              (rest (cdr lst)))
          (if (gap? $elt)
              (call/down-slag accum rest)
              (combiner $elt
               (future (call/down-up-slag
                                accum rest)))))))
  (call/down-up-slag accum sll))
```

Figure 5: OPERA implementations of some simple synchronized lazy list operators.

`call/down-slag` in this case. This accounts for the missing `up[A,2]` line in the test of `up-average` in Section 4.1.

While the above presentation gives the flavor of slag operators, it glosses over a number of subtleties. It turns out that handling filtering in a fully reusable manner requires slags to carry extra information per element and synchrons to have parent/child relationships with special garbage collection properties. Also, writing certain kinds of transducers can be much trickier than the above examples indicate. [Tur94] describes these complexities in detail.

5 Experience

We have implemented a prototype OPERA interpreter based on an explicit-demand graph rewriting model (see [Tur94] for details). Fine-grained concurrency, synchrons, lazons, and futures are relatively straightforward to handle in this model. The graph model facilitates a reference-counting implementation of barrier synchronization; synchronization only requires a full-fledged garbage collection in the rare case when a synchron is held in a cyclic fashion by a stale non-waiting pointer.

Using our interpreter, we have tested a suite of higher order SLL and SLT procedures on hundreds of simple examples. Our examples include operators that perform mutation and networks that exhibit cycles and loose coupling. These preliminary tests confirm our expectation that slags are a worthwhile technique for creating reusable components whose combinations exhibit desirable time and space profiles.

Experience with our interpreter indicates that high time overheads are incurred by the concurrent evaluation mechanism, synchronization, and the management of slags. It remains a challenge to reduce these overheads or eliminate them by static analysis.

We have found that it can be challenging to write slag operators that obey all of the operator requirements listed in Section 3.4. A common problem is spurious deadlocks resulting from synchron references that are not aggressively unbundled. We have developed the DYNAMATOR, a graph-based program animator, to help us find the source of deadlocks. The DYNAMATOR has been an invaluable tool for constructing slag operators that behave as expected.

In their current form, we believe that synchrons are too subtle to useful to the average programmer. Instead, we imagine that they will usually be packaged into abstractions (e.g., higher order operators on synchronized lazy aggregates) that are carefully crafted by experts.

We are intrigued by the notion that slag operators can be characterized by a *shape* attribute that summarizes interactions between data dependencies and patterns of recursive calls. For example, SLL operators can be characterized as *down* (iterative), *up* (non-iterative), or *across* (work in both iterative and non-iterative contexts). SLT operators support a much richer shape vocabulary. Our experience with slag operators suggests the existence of a shape calculus for determining the shape of a slag component from the shapes of its operators.

6 Conclusion

We have described a dynamic model for the lock step processing of programs written in an aggregate data style. The model enables programmers to simulate desirable operational aspects of monolithic computations (especially space profiles) in a modular fashion. The model can be viewed as a dynamic version of Waters's series package that handles general linear and tree-shaped recursions in addition to iterations. The key feature of the model is the synchron, a novel barrier synchronization mechanism that interacts with automatic storage management. Synchrons enable the barrier represented by a strict procedure call to be distributed across operators that communicate via synchronized lazy aggregates. When combined with fine-grained concurrency, synchrons enable the lock step processing of slag operators.

The work described here is one step towards the holy grail of efficient modular programs. It can be extended in a number of important ways:

- *Pragmatics*: In order for slags to be a practical technique, it is necessary to reduce various overheads exhibited by the current implementation. The *dynamic* approach is to develop more efficient concurrency and synchronization mechanisms. The *static* approach is to eliminate concurrency, synchronization, and data manipulation overheads by compiling slag networks into monolithic recursions. We expect that the techniques for compiling series [Wat91] can be extended to a restricted subset of slag networks, and that the explicit synchronization information available in slags can facilitate their compilation.

- *Shape analysis*: In analogy with type systems, the notion of shape introduce above suggests a *shape system* in which the shape of a network can be determined from the shape of its components by *shape reconstruction*. Shape analysis would be helpful for conservatively predicting deadlock in a slag network. Also, any slag compilation technique would presumably have to employ some sort of shape analysis.

- *Formalizing time and space profiles*: The notions of time and space profiles introduced above are informal. However, we believe that the pomset model described in [HA87] can be extended to make these notions precise.

- *Pedagogy*: We believe that the computational models developed for this work can have important pedagogical applications because they provide alternate approaches for reasoning about the structure of monolithic recursions. In fact, the most important contribution of this work is that it suggests new ways to think about computation.

Acknowledgments

The research described here is part of a doctoral dissertation completed under the supervision of Gerry Sussman and David Gifford, with lots of help from Dick Waters. Jonathan Rees and Jim O'Toole suggested major structural improvements to this paper. Brian Reistad, Mark Sheldon, Trevor Jim, and David Espinosa provided valuable feedback on the paper.

This paper describes research done at the Artificial Intelligence Laboratory and the Laboratory for Computer Science at the Massachusetts Institute of Technology. Support for this research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097, by the National Science Foundation under grant number MIP-9001651, and by the Department of the Army under contract DABT63-92-C-0012.

References

[Ada91] Stephen Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, Department of Electronics and Computer Science, University of Southampton, March 1991.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, pages 598–632, October 1989.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[Axe86] Tim S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Computing*, 3(2):129–140, May 1986.

[Bar92] Paul S. Barth. Atomic data structures for parallel computing. Technical Report MIT/LCS/TR-532, MIT Laboratory for Computer Science, March 1992.

[Bel84] Francoise Bellegarde. Rewriting systems on FP expressions that reduce the number of sequences they yield. In *Symposium on Lisp and Functional Programming*, pages 63–73. ACM, August 1984.

[Bir89] Andrew Birrel. An introduction to programming with threads. SRC Report 35, Digital Equipment Corporation, January 1989.

[CM90] Eric Cooper and J. Gregory Morrisett. Adding threads to standard

ML. Technical Report CMU-CS-90-186, Carnegie Mellon University Computer Science Department, December 1990.

[CR+91] William Clinger, Jonathan Rees, et al. Revised[4] report on the algorithmic language Scheme. *Lisp Pointers*, 4(3), 1991.

[DC90] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, April 1990.

[Den75] Jack B. Dennis. First version of a data flow procedure language. Computation Structure Group Memo MIT/LCS/TM-61, MIT Laboratory for Computer Science, May 1975.

[FMY92] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implmentation. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 223–234, 1992.

[HA87] Paul Hudak and Steve Anderson. Pomset interpretation of parallel functional programs. In *Functional Programming Languages and Computer Architecture*, pages 234–256, September 1987. Lecture Notes in Computer Science, Number 274.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, pages 501–528, October 1985.

[HJW+92] Paul Hudak, Simon Petyon Jones, Philip Wadler, et al. Report on the programming language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hug83] R. J. M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Oxford Universiy Computing Laboratory, Programming Research Group, July 1983.

[Hug84] R. J. M. Hughes. Parallel functional languages use less space. Technical report, Oxford University Programming Research Group, 1984.

[Hug90] R. J. M. Hughes. Why functional programming matters. In David Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.

[Ive87] Kenneth E. Iverson. A dictionary of APL. *APL QUOTE QUAD*, 18(1):5–40, September 1987.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987. Lecture Notes in Computer Science, Number 274.

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[L+79] Barbara Liskov et al. CLU reference manual. Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, October 1979.

[Mil87] James S. Miller. MultiScheme: A parallel processing system based on MIT Scheme. Technical Report MIT/LCS/TR-402, MIT Laboratory for Computer Science, September 1987.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Ste77] Guy L. Steele Jr. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or LAMBDA, the ultimate

Goto. Technical Report AIM-443, MIT Artificial Intelligence Laboratory, October 1977.

[Tur94] Franklyn Albin Turbak. *Slivers: Computational Modularity via Synchronized Lazy Aggregates (in preparation).* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1994.

[WA85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *2nd European Symposium on Programming*, pages 344–358, 1988. Lecture Notes in Computer Science, Number 300.

[Wat90] Richard C. Waters. Series. In Guy L. Steele Jr., editor, *Common Lisp: The Language*, pages 923–955. Digital Press, 1990.

[Wat91] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.