Chapter 1

Overview

Slivers are a new approach to expressing computations as combinations of mix-and-match operators on aggregate data. Unlike other aggregate data models, slivers enable programmers to control fine-grained operational aspects of modular programs. In particular, slivers can guarantee that networks of operators exhibit the desirable storage behavior and operation scheduling of intricate loops and recursions. For example, slivers can preserve the space efficiency of a complex tree algorithm when it is expressed as the superposition of simpler tree walks.

The sliver technique is based on a dynamic model of lock step processing that enables combinations of list and tree operators to simulate the operational behavior of a single recursive procedure. Operational control is achieved through *synchronized lazy aggregates*, dynamically unfolding data structures that constrain how the processing of separate operators is interwoven. The key to the technique is the *synchron*, a novel first-class object that allows a dynamically determined number of concurrently executing operators to participate in a barrier synchronization. Slivers embody a notion of *computational shape* that specifies how the operational patterns of a process can be composed out of the patterns of its components.

The utility of slivers is illustrated in the context of SYNAPSE, a simple language for expressing linear and tree-shaped computations. SYNAPSE is built on top of OPERA, a new concurrent dialect of Scheme that incorporates the concurrency, synchronization, and non-strictness required by the lock step processing model. The semantics of OPERA are explained in terms of EDGAR, a novel graph reduction model based on explicit demand propagation.

1.1 The Problem

Ideally, programming languages should encourage programmers to express their designs in a modular fashion based on a library 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. They manually interweave common processing idioms into *monolithic* programs that do not exhibit the modular nature of the idioms.

In this research, I investigate the problem of decomposing programs into mix-and-match parts that preserve the operational character of the original programs. I focus in particular on decomposing loops and recursions. For instance, consider a single loop that computes both the sum and the length of a numeric sequence. We would like to express such a loop as the composition of two loops, one of which computes the sum of a sequence and the other of which computes the length of a sequence. Similarly, consider decomposing a complex single-traversal tree walk into separate components that propagate information top-down, bottom-up, and left-to-right. We would like the resulting modular program to perform a single tree traversal with the same time and space requirements as the original program.

While numerous techniques have been developed for factoring loops and recursions into modular components, most fail to preserve operational properties like time and space requirements. Whereas a monolithic single-traversal tree walk often requires space proportional to the depth of the tree, it is common for the 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, the modularity techniques are unduly restricting the class of computations that the programmer can describe.

The tree walk example illustrates the two-edged sword of modularity. On the one hand, modularity simplifies program design and modification by hiding all sorts of details behind the narrow interfaces of components that are simple to reason about and combine. On the other hand, modularity necessarily prevents the user of the components from controlling the hidden details to improve behavior. The trick of good component design is to ensure that the interface is wide enough to allow desirable behavior, but not so wide as to overwhelm the user and overconstrain the implementer.

My thesis is that existing techniques for modularizing loops and recursions unnecessarily prevent the programmer from controlling the operational behavior of a network of components. In this dissertation, I develop an alternate technique, sliver decomposition, for breaking loops and recursions into components with operationally desirable composition properties. The key to sliver decomposition is widening the interface of traditional components to include important synchronization control points. These control points enable programmers to better express how a network of components should behave.

In the remainder of this section, I motivate issues of modularity and control addressed by this work, and summarize the drawbacks of existing mechanisms for modularizing monolithic loops and recursions.

1.1.1 Modularity: Programming Idioms

Programs are rarely written "from scratch". Existing code often serves as a template that can be molded into a desired program component. Library routines free the programmer from reimplementing common functionality. But even in the case where programmers eschew existing code and library routines, they almost always make heavy use of programming patterns they have seen or written before. These common patterns of usage, often called *idioms* or *cliches*, are central to programming. Recognizing idioms when reading code and effortlessly integrating idioms when writing code are key abilities that distinguish expert programmers from run-of-the-mill programmers.

Idioms are often not highlighted in the program text. For example, consider a running sum idiom, in which a variable is first initialized to 0, and then is updated during a computation so that it holds a running total of certain numbers generated during the computation.

The declaration, initialization, and updating of the variable are typically spread throughout the program text, rather than being localized to a single region. The distributed nature of typical idioms makes them difficult for program readers to find and for program writers to keep in their heads.

Programming environments can help to programmers to manage idioms. One approach is to provide tools, intelligent assistants, and special-purpose languages that aid the programmer in analyzing and synthesizing programs in terms of idioms. A good example of this approach is Rich and Waters's Programmer's Apprentice project [RW90]. An alternate approach to idiom management is to devise language constructs and mechanisms for encapsulating idioms as single entities within a general-purpose programming language. This is a basic motivation for modularity and abstraction in programming languages, and is the approach taken here. In particular, I will focus on techniques for capturing idioms that occur in loops and general recursive programs. Such idioms include components that generate, filter, map, and accumulate sequences and trees. While a program might not explicitly manipulate list-structured or tree-structured data, the time-dependent values of variables manipulated by the computation often naturally exhibit the structure of a sequence or tree that unfolds over time.

1.1.2 Control: Computational Shape

Programming would be a much simpler task if all that mattered about a program was that it had the correct input/output behavior. In practice, programmers care a great deal about *how* the outputs are computed from the inputs. They select algorithms and use programming styles that make effective use of various resources. Sometimes this means reducing program execution time and space or improving hardware utilitization. Other times it means writing code that is quick to implement, easy to read, simple to maintain, or decomposable in ways that effectively use the talents of all members of a programming team.

I assume throughout this dissertation that an essential aspect of programming is controlling the way that computational processes unfold over time. Details of process evolution determine the machine-based resources, such as time and space, required by a program. Even higher-level properties like program readability, writability, and modifiability are closely tied to patterns of process evolution. Process patterns common enough to be considered idioms are more easily programmed and recognized than idiosyncratic ones.

Following Abelson and Sussman [ASS85], I refer to patterns of process evolution as *computational shapes*. Some examples of computational shapes are iterations in two state variables, linear recursions, and left-to-right pre-order tree walks. Intuitively, a computational shape captures operational details like the relative ordering of operations and the storage profile of a process during its evolution.

I also adopt the view set forth by Abelson and Sussman that a central activity of programming is designing descriptions that give rise to imagined patterns of process evolution. From this perspective, programmers are process potters who mold computational clay into desired shapes. A goal of this dissertation is to support this view of programming by capturing idiomatic process patterns as programming language entities that compose in the "right way". Along the way, I will explain why existing techniques for expressing these idioms fail to have satisfactory composability properties.

1.1.3 The Signal Processing Style of Programming

A natural way to decompose monolithic loops and recursions is to view them as signal processing systems in which information flows through devices that generate, map, filter, and accumulate data. I call this method of structuring programs the *signal processing style* (SPS) of programming.

As a motivation for SPS, consider a program that lists the minimum, maximum, and average of the salaries of active employees from a given employee database. A modular approach is to write separate subprograms that accumulate the minimum, maximum, and average of any sequence of numbers, and then hook these components up to another component that generates a sequence of salaries of active employees from the database. This approach is graphically depicted in Figure 1.1 by the box labelled `SALARY-INFO`.

Figure 1.1 also illustrates the hierarchical nature of this programming style. The salary generator can itself be factored into parts that generate a sequence of records from a database (`GENERATE-RECORDS`), weed out the records of inactive employees (`FILTER-ACTIVE`),

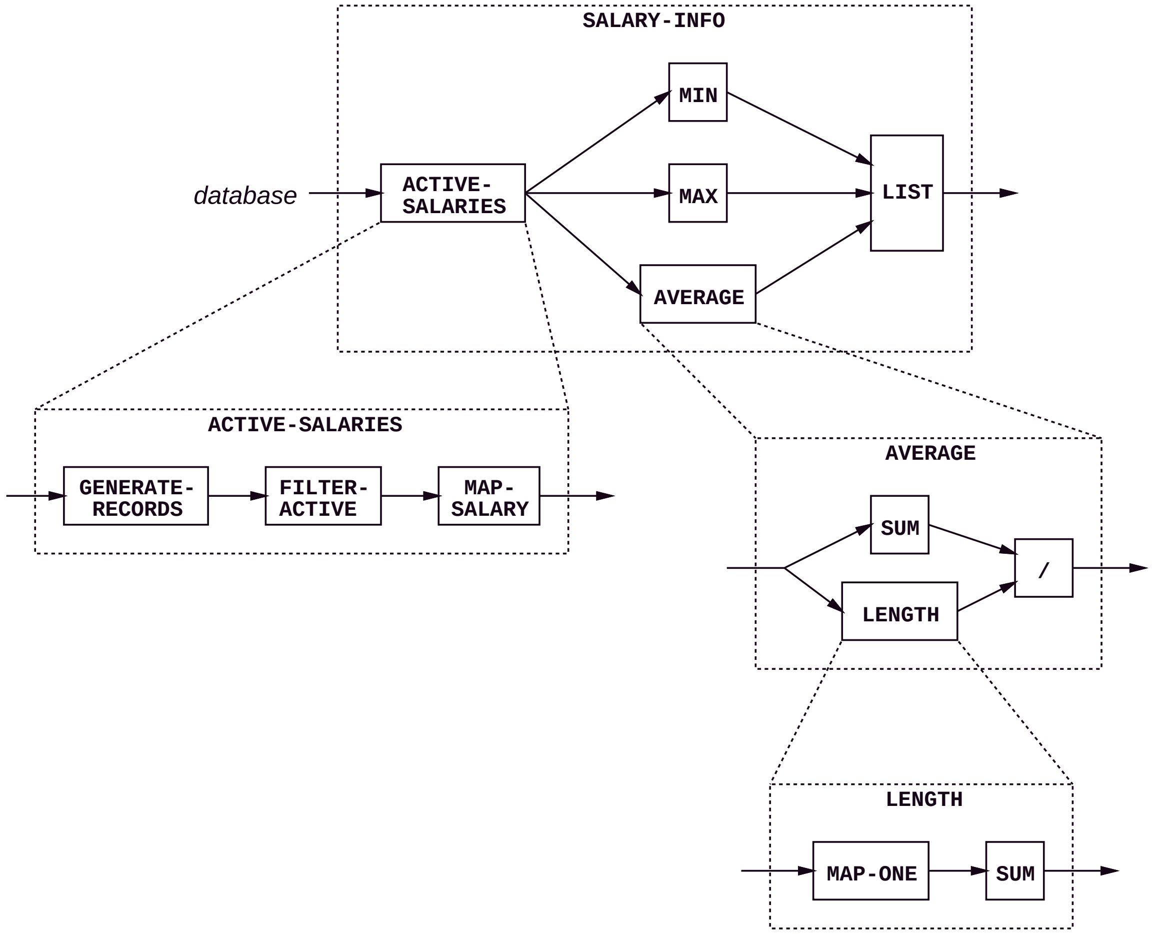


Figure 1.1: Structure of a modular program for computing the minimum, maximum, and average salaries for active employees from a given database.

and extract the salary from every remaining record (`MAP-SALARY`). The averaging subprogram can be expressed as the quotient of two other components, one that calculates the sum of a numeric sequence (`SUM`) and another that calculates the length of any sequence (`LENGTH`). Even the `LENGTH` component admits a decomposition into a sum of a sequence of ones that is the same length as the input sequence. Because the parts depicted in the figure are not only reusable but applicable in a wide range of contexts, they constitute the basis of a powerful modular programming language.

The signal processing style of programming has a long history and is supported by numerous mechanisms in a wide range of programming languages. This style was used at least as far back as the early 1960's in the form APL's array operators and Lisp's higher-order list manipulation procedures. Today, the signal processing style is supported by a variety of mechanisms that I broadly classify into two approaches:

1. The *aggregate data approach* treats devices as operators that manipulate aggregate data structures like lists, trees, and arrays. This approach includes functions on strict and lazy data and data-parallel vector operators.

2. The *channel approach* treats devices as processes that communicate via some sort of data channel. This approach includes communicating threads, file-processing pipes, producer/consumer coroutines, and dataflow techniques.

The sliver technique introduced in this dissertation augments the aggregate data approach but mixes in some crucial features from the channel approach.

Existing SPS techniques suffer from various drawbacks that limit the range of computations that they can express:

- *Limited Shapes*: In many SPS frameworks, devices are constrained to process a linear stream of information in an iterative fashion. While linear iterative computations are perhaps the most common computational shape, this limitation excludes more general linear recursions and all tree recursions.[1]

[1] 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.

- *Constrained Topologies*: Some SPS techniques circumscribe the ways in which devices can be connected. Many mechanisms in the channel approach only allow *straight-line* networks — i.e., linear networks in which the output of each device can be connected to the input of only one other device. However, many programs (`SALARY-INFO`, for example) decompose into networks that exhibit *fan-in* (where a device consumes multiple inputs) and *fan-out* (where a device output is used in multiple places). Another common restriction prohibits *cyclic paths* from a device output back to one of its inputs. Yet, some programs naturally decompose into standard parts connected by cyclic paths.

- *Excessive Space Overhead*: SPS programs can require significantly more space than their monolithic counterparts. Consider the `SALARY-INFO` program described above. If we assume that the records in the employee database are linearly accessible, then it is easy to write the salary program as a monolithic loop that uses only constant space. Yet, almost all aggregate data mechanisms, as well as channel-based mechanisms that do not support fan-out, lead to modular programs that require space proportional to the size of the database!

  As explained in Chapter 3, even techniques like laziness and fancy program transformations do not ameliorate this storage disaster in general. The problem has its roots in a fundamental mismatch between *demand-driven evaluation* and the *fan-out* of results from device outputs. As far as I know, only Waters [Wat91] and Hughes [Hug83, Hug84] have provided partial solutions to this problem within aggregate data approach. (In channel-based mechanisms that support fan-out, the space problem is often solved by bounded channels.)

- *Excessive Time Overhead*: For a variety of reasons, SPS programs can require significantly more time than their monolithic counterparts. Some of the time overhead is due to the manipulation of intermediate aggregates or channels that are not present in the monolithic version. I consider this overhead acceptable in the sense that it is a reasonable cost for the benefits achieved by modularity. Furthermore, this overhead can often be reduced by clever compilation strategies.

On the other hand, time penalties due to a mechanism's lack of expressive power are unreasonable. For example, when using a mechanism that does not support fan-out, it is necessary to replicate devices whose results are used in more than one place (see Figure 1.2); this leads to an unnecessary duplication of work. As another example,

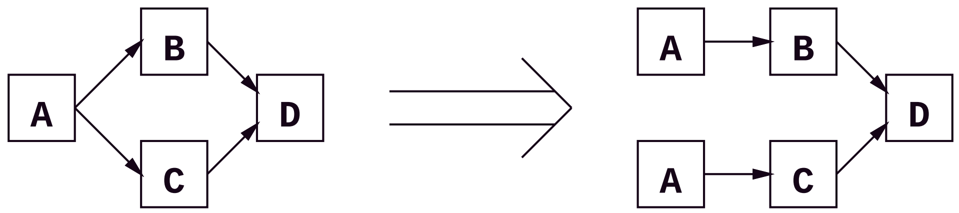


Figure 1.2: In SPS techniques that do not allow fan-out, a device must be replicated.

mechanisms not supporting a delayed evaluation strategy can perform unnecessary work. Consider the network in Figure 1.3; if the `EVERY-OTHER` device is a filter that passes only the even-indexed elements, then the `MAP-EXPENSIVE-FUNCTION` should ideally not perform any computation on the odd-indexed elements. (A corresponding monolithic program would almost certainly avoid these unnecessary computations.)

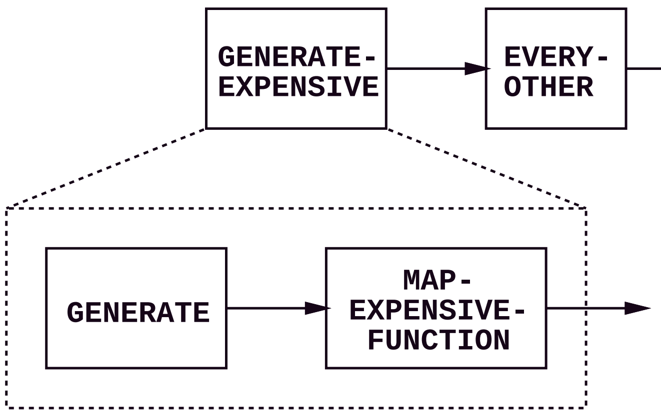


Figure 1.3: A network for which many SPS techniques perform unnecessary work.

1.2 Sliver Decomposition

Sliver decomposition is a new technique for modularizing loops and recursions. It enhances the expressive power of SPS programming by ameliorating the problems of exisiting techniques outlined above.

1.2.1 The Basic Idea

Sliver decomposition augments the aggregate data approach by extending the operators and aggregates to handle a simple notion of computational shape. Shape is encoded in the way that the operators interact with a shared synchronization structure communicated by the aggregates. An abstract depiction of sliver decomposition appears in Figure 1.4. Here,

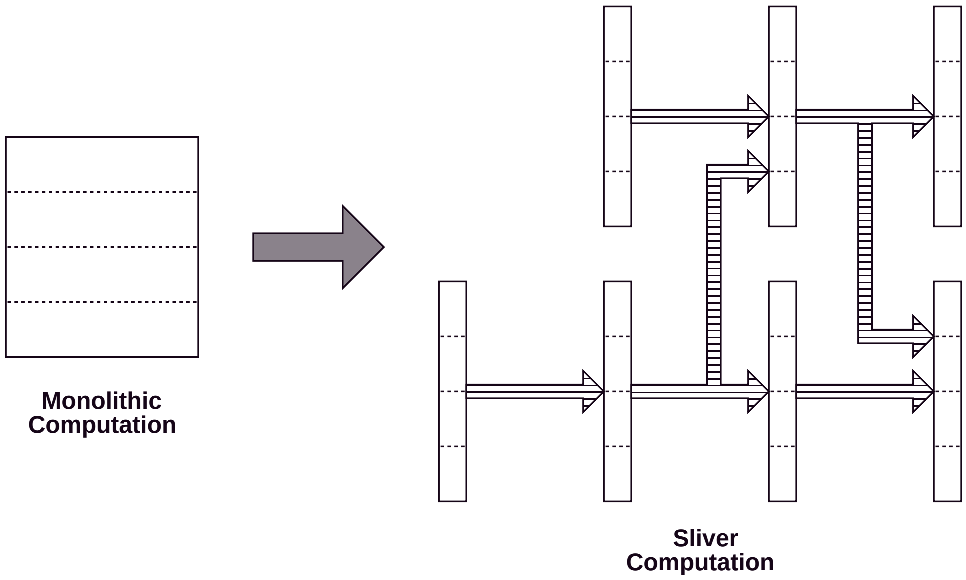


Figure 1.4: Decomposing a monolithic computation into a network of slivers.

a monolithic recursive computation is partitioned into a network of *slivers* (tall skinny boxes) that communicate via *synchronized lazy aggregates*, or *slags* (thick arrows). The major difference between this *sliver diagram* and the other SPS diagrams encountered so far is that the devices and wires exhibit some structure. Each horizontal dotted line is a *call boundary* that represents one of the recursive calls made in a recursive computation.

The area between two such lines represents the computation performed by one level of the recursion. The decomposition distributes the recursive call structure of the monolithic computation across each of the slivers. The striations of the slags are intended to suggest that they transmit a representation of the recursive call structure of one sliver to another.

A sliver network resulting from sliver decomposition is intended to satisfy two criteria:

1. *Operational faithfulness*: The network as a whole should preserve the operational behavior of the monolithic computation. Here, operational behavior includes which operations are performed by the monolithic computation, the relative order of these operations, and the storage profile of the whole computation. The network is allowed to employ additional operations and storage for management purposes as long as it maintains the monolithic computation's order of growth in space and time.

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

Reusability is achieved by representing slivers as procedures and slags as data structures. These choices mean that sliver networks can be expressed by standard mechanisms for procedural composition.

Operational faithfulness is achieved by a lock step processing model that guarantees that corresponding call boundaries of the individual slivers are glued together to simulate a call boundary of the monolithic computation. This gluing process is depicted in Figure 1.5. Communication events (arrows labelled C) occur between pairs of connected slivers, but synchronization events (shaded barriers labelled S) are shared among all the slivers. The idea is that every sliver computation locally must wait at the shared barrier until all the other slivers in the network have reached the same barrier. By tightly coupling the sliver computations, the synchronization barriers propagated by the slags ensure that the network as a whole behaves like a monolithic procedure.

In this context, "shape" describes the time-based relationships between the synchronization barriers and how the slivers interact with these barriers. Each barrier is actually associated with *two* events in a computation: calling a recursive procedure and returning from a recursive procedure. A computation can be viewed as a path that crosses each bar-

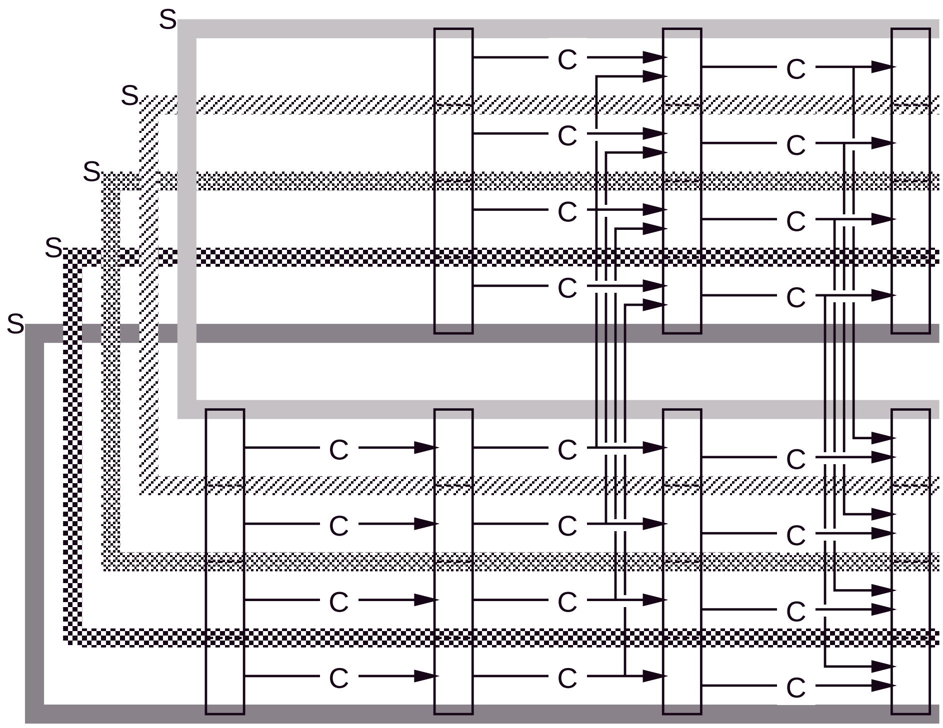
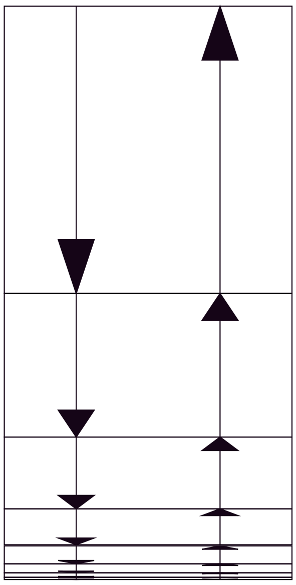
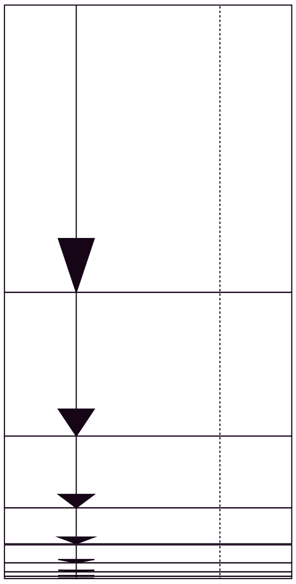


Figure 1.5: Communication events (C) and synchronization events (S) among the slivers in a network.



(a) Shape of a general linear recursion.

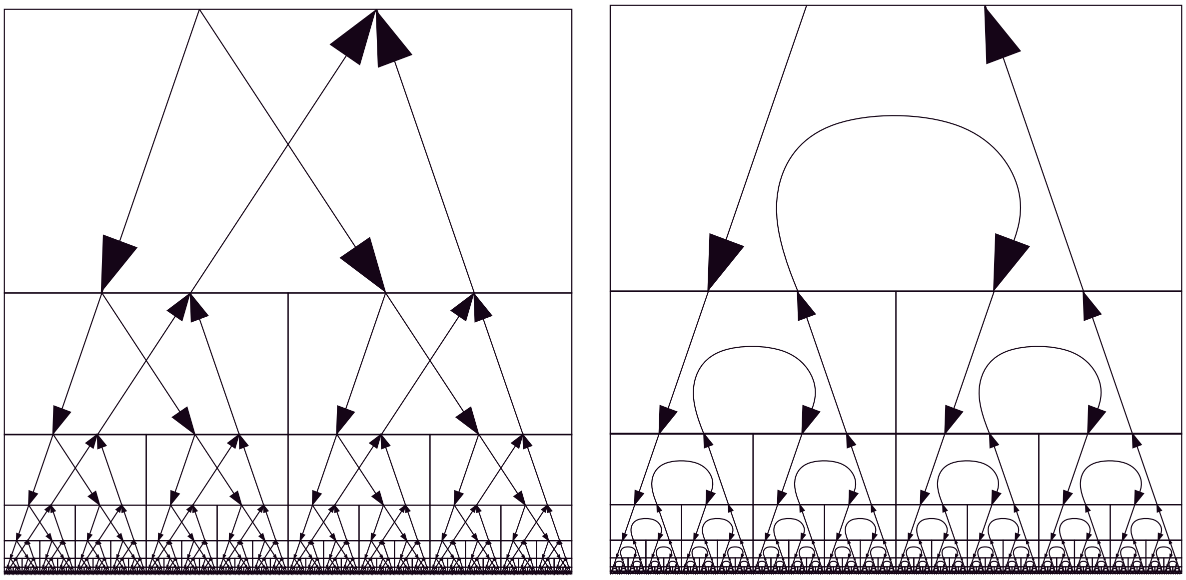


(b) Shape of a linear iteration.

Figure 1.6: Some shapes for linear computations. Each horizontal line is a synchronization barrier whose left half represents a call event and whose right half represents a return event. Solid directed lines indicate a time ordering between events, while undirected dotted lines connect simultaneous events. A chain of downward arrows represents the iterative (calling) portion of a linear recursion. A chain of upward arrows represents the recursive (returning) portion of a linear recursion. A chain of undirected dotted lines represents the non-returning behavior of tail calls.

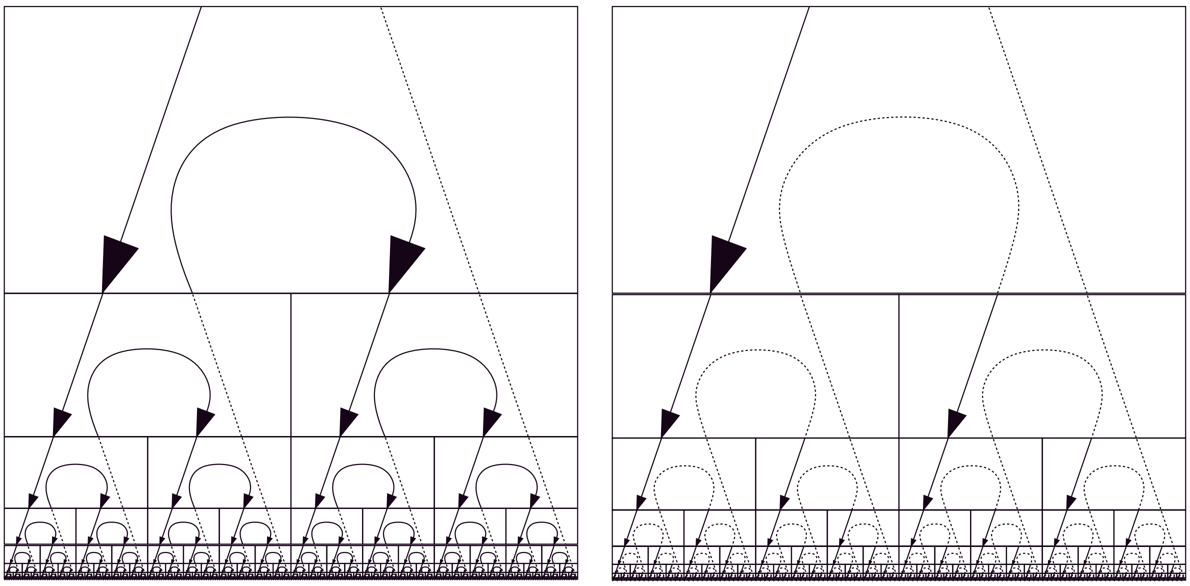
rier in a synchronization structure twice: once for the call event, and once for the return event. For example, Figure 1.6(**a**) is an abstract depiction of a general linear recursion. Such a computation breaks cleanly into a down (call) phase and and up (return) phase. An iterative linear computation is a special case in which each call is a non-returning *tail call* [Ste77]; it exhibits no up phase, because all returns events effectively occur at the same time (as indicated by the dotted lines in Figure 1.6(**b**)). These notions extend to tree computations; Figure 1.7 is a gallery of some shapes for binary computation trees.

The shape of the computation defined by a network is derived from the shapes of its component slivers. For example, if each linear sliver in a network is iterative, then the network as a whole is iterative. But if one sliver has a non-trivial up phase, then so does the whole network. Tree-shaped computations permit a wider and more interesting variety



(a) Parallel tree computation.

(b) Post-order tree computation.



(c) In-order tree computation.

(d) Pre-order tree computation.

Figure 1.7: Some common shapes on a binary computation tree. Each call/return barrier lies above the barriers for its two recursive subcalls. Shape **(a)** is the multi-threaded walk of a parallel tree computation, while shapes **(b)**-**(d)** are variations on single-threaded left-to-right walks of a sequential tree computation. Other binary shapes include right-to-left versions of the left-to-right shapes.

of shape combinations.

The fact that each sliver network corresponds to single recursive computation constrains the kinds sliver combinations that make sense. There is a kind of shape calculus on slivers that determines the compatibility of the slivers in a network. In a network of linear slivers, for instance, the down phase of a sliver may consume the products of a preceding sliver's down phase but not those of its up phase; the latter situation would not correspond to a single-pass linear recursion. Some rules for binary computations are that parallel slivers usually mix with the sequential ones, but left-to-right and right-to-left binary shapes are always incompatible.

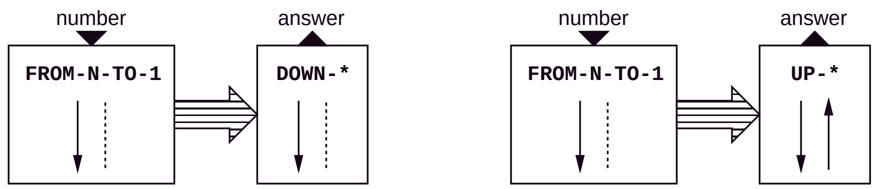
Sliver decomposition is intended not to replace other SPS techniques, but to be used in conjunction with them. The lock step processing of sliver network is not appropriate for many computations. However, sliver decomposition interfaces nicely with other aggregate data mechanisms, so it is easy to flexibly mix the tight coupling of slivers with the loose coupling afforded by other mechanisms.

1.2.2 Some Simple Examples

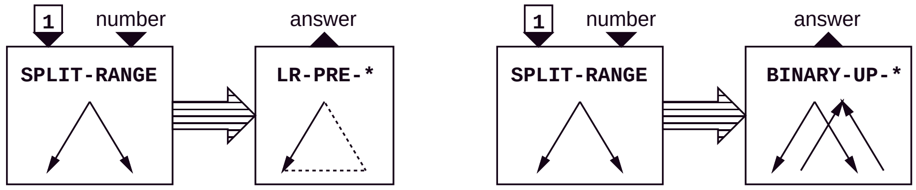
In this section, I present a few simple examples that give the flavor of sliver decomposition and hint at its expressive power. (The examples in this section are necessarily brief and simple. The reader is encouraged to explore the more interesting examples in Chapter 6.)

First consider the time-worn, but still trusty, factorial procedure. A procedure for calculating the factorial of n naturally breaks into two parts: a generator of the numbers between 1 and n, and an accumulator that takes the product of these numbers. The sliver diagrams in Figure 1.8 illustrate that this decomposition is supported by many different computational shapes.

In **(a)**, the `FROM-N-TO-1` sliver generates the integers from the input down to (and including) 1, while the `DOWN-*` iteratively accumulates these numbers. The downward arrows annotating the slivers and the "`DOWN`" in `DOWN-*` indicate that both slivers have only a down phase, so the resulting computation is a linear iteration. In contrast, the `UP-*` accumulator of **(b)** has an up arrow because it stacks multiplication operations to be performed after the last number is generated; the resulting computation is a non-iterative recursion. In both



(a) Linear iterative factorial. **(b)** Linear recursive factorial.



(c) Left-to-right pre-order tree factorial. **(d)** Parallel tree factorial.

Figure 1.8: Various sliver decompositions of a factorial procedure.

(a) and **(b)**, it would be possible to replace the generator by a `FROM-1-TO-N` sliver that counted from 1 up to the input. This would yield two more operationally distinct versions of factorial.

Factorial versions **(c)** and **(d)** describe tree-shaped computations. In both, the `SPLIT-RANGE` generator takes a range specified by low and high bounds and creates a binary tree slag whose leaves are the numbers in the range. Given a range that contains only a single element, `SPLIT-RANGE` produces a leaf with that element; otherwise it produces a valueless tree node whose left and right subtrees are trees for two balanced subranges that partition the given range. The generator has a so-called "binary down" shape because range information conceptually travels in parallel from a parent node down to both subnodes.

In **(c)**, the product of the leaves is calculated by the left-to-right pre-order `LR-PRE-*` accumulator, while in **(d)**, the subtree products of the subtrees are conceptually evaluated in parallel and then combined by the `BINARY-UP-*` accumulator. Due to the operational faithfulness of slivers, the computation described by version **(c)** uses control space proportional to the depth of the tree; at most one branch of the tree really exists at any point in time. However, in **(d)**, the multi-threaded nature of a parallel computation implies that

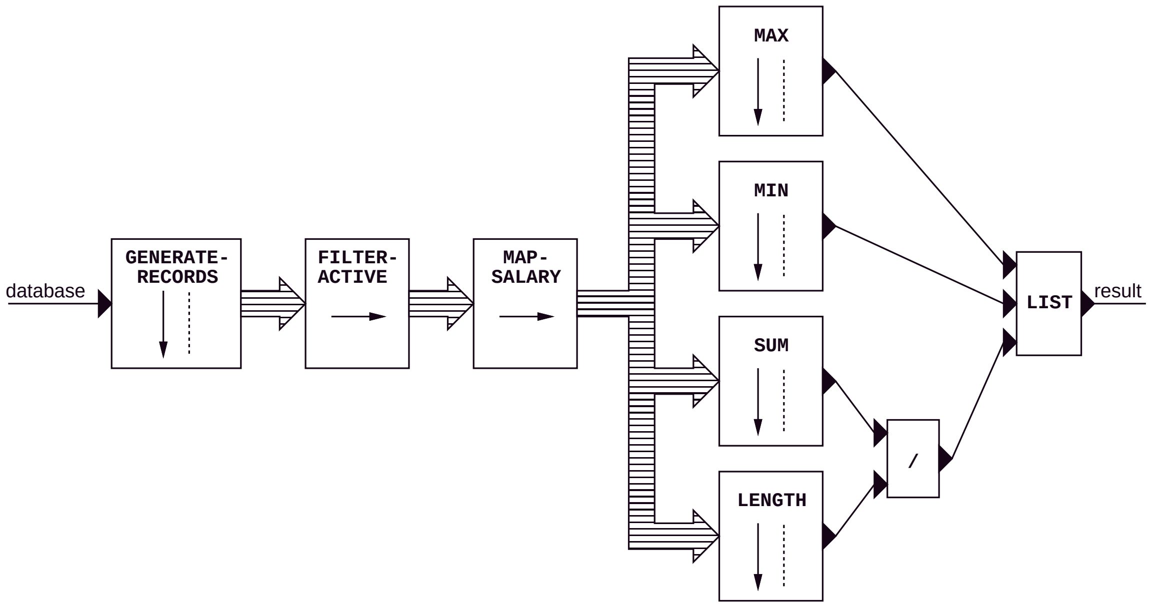


Figure 1.9: The salary information program expressed as a sliver diagram. The thick cables represent slags, while the thin lines represent non-slag data. The whole network behaves like a monolithic iteration because each of the components is inherently iterative.

space proportional to the size of the whole tree may be required in the worst case. Of course, there are many other strategies for generating a tree of numbers and finding their product. The shape-based nature of slivers makes them a good language for describing and comparing various approaches to a problem.

Figure 1.9 presents a sliver diagram for the salary information program presented earlier. Because all elements of the network have a down shape, the specified computation is guaranteed to behave like a monolithic iteration in five state variables (current record, current minimum, current maximum, current sum, and current count). This is an important improvement over SPS techniques that disallow fan-out or would build up intermediate storage proportional to the size of the list. Replacing any one of the slivers by a component with up shape would specify a computation requiring a linear stack.

The sliver diagram in Figure 1.10 exercises some of the other kinds of linear slivers that can be expressed:

- `SPLAY-LIST` converts a list into a linear slag.

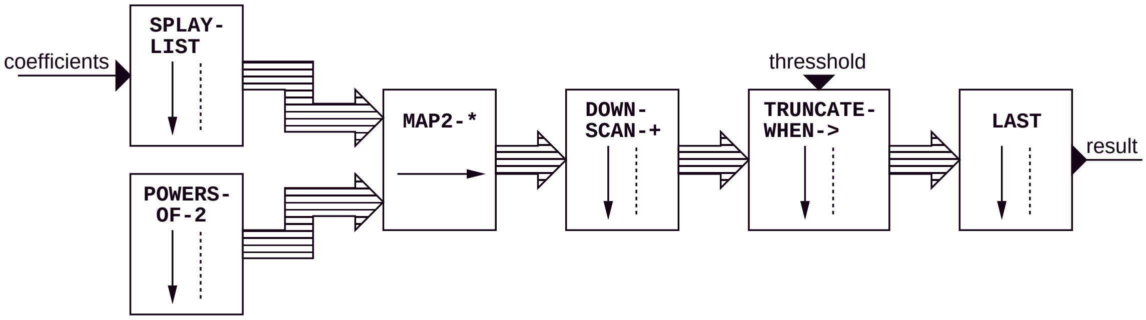


Figure 1.10: A sliver diagram introducing some new kinds of linear slivers.

- `POWERS-OF-2` generates a conceptually infinite slag with elements 2^0, 2^1, 2^2, ...
- `MAP2-*` is a two-input mapper that performs elementwise multiplication. Its output is only as long as its shortest input.
- `DOWN-SCAN-+` performs an iterative sum accumulation, but returns a slag of the intermediate sums rather than just the final answer.
- `TRUNCATE-WHEN->` truncates the input slag after the first element greater than a given threshold.
- `LAST` returns the last element of a given slag.

The program as a whole iteratively calculates the first sum in a running sum of scaled powers of two that is greater than a particular threshold.

Finally, consider some simple tree examples. Figure 1.11 shows three tree slivers that transform one tree-shaped slag into another. Each node of a tree slag is assumed hold a number. Each of the slivers returns a new tree slag in which every node is annotated with the intermediate sum maintained by a particular tree summation computation when it processes the node. `BINARY-DOWN-SCAN-+` returns at each node the sum of the numbers on the direct path to the root; `BINARY-UP-SCAN-+` returns at each node the sum of the numbers in the subtree rooted at that node; and `LR-PRE-SCAN-+` returns at each node the running sum of a left-to-right pre-order summation. Following the terminology from data-parallel programming, we refer to these slivers as *scanners*.

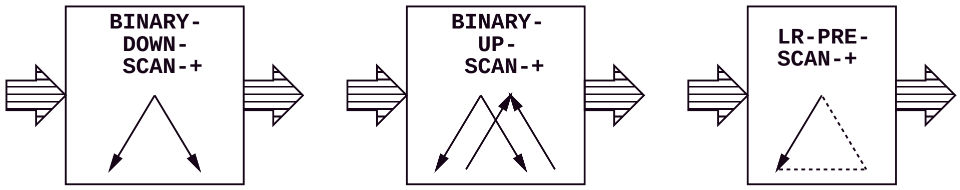


Figure 1.11: Three tree scanners distinquished by shape.

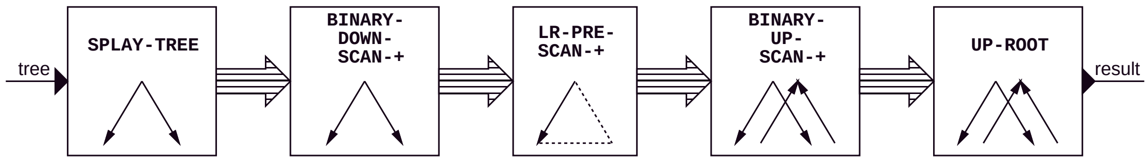


Figure 1.12: A sequential composition of differently shaped tree scanners.

The tree scanners can be combined both sequentially and in parallel. For example, Figure 1.12 returns the root value of the tree resulting from a sequential cascading of these three scanners, while Figure 1.13 returns the maximum value of a given function applied elementwise to a given tree and the results of the three scans on that tree. In both cases, the computation described by the sliver diagram behaves like the single-traversal tree walk of a corresponding monolithic recursive procedure.

Although these tree examples are contrived, the shapes involved suggest more practical applications. Tree slivers can be used to manipulate tree-structured databases and abstract syntax trees of programs. This makes it possible to express such programs as pattern-matchers, deductive retrievers, interpreters, and compilers as networks of slivers.

1.2.3 How it Works

Sliver decomposition makes essential use of concurrency, synchronization, and laziness:

- *Concurrency*: The demand-driven model underlying sliver decomposition is inherently concurrent. The interaction between demand-driven evaluation and fan-out requires some form of concurrency to prevent spurious storage leaks (see [Hug84]). Experience with sliver decomposition suggests that concurrency is an essential technique for

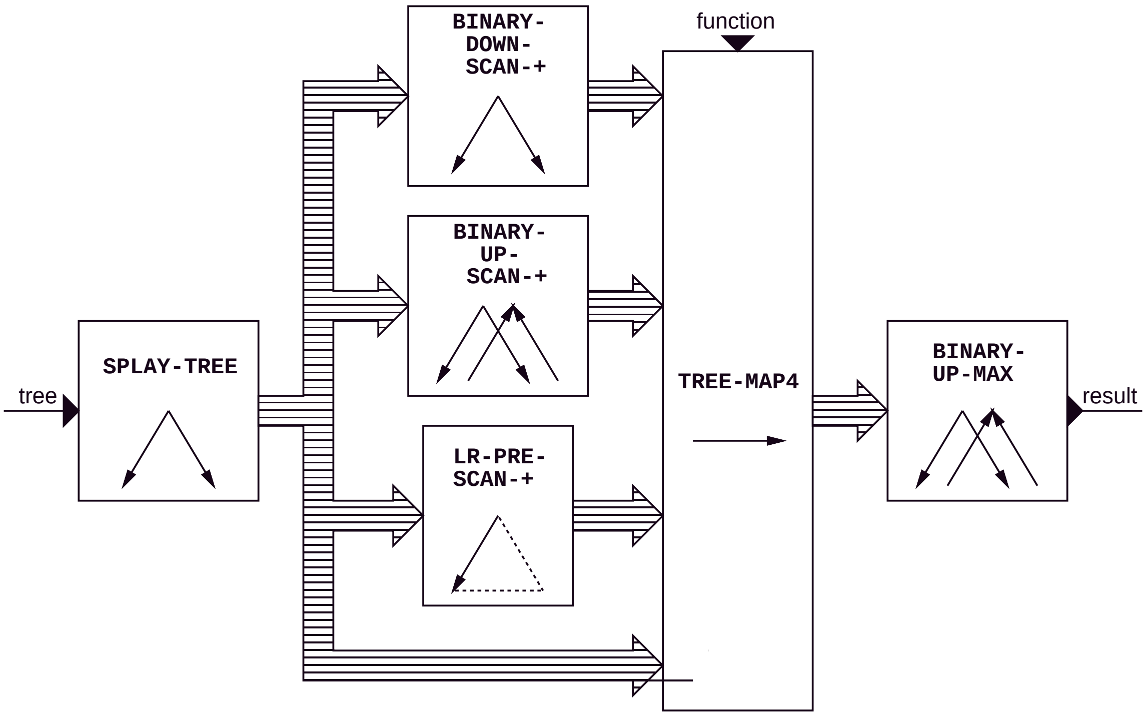


Figure 1.13: A parallel composition of differently shaped tree scanners.

expressing programs in a modular fashion.

- *Synchronization*: The lock step processing of sliver networks is achieved by *synchrons*, a novel synchronization technology. A sliver network dynamically "solves" a set of time constraints between sliver operations and the call or return events represented by a set of synchrons. An event represented by a synchron is only enabled when the system can "prove" that no more computation can happen before the event. Locks, a more traditional forms of synchronization for concurrent systems, are also supported.

- *Laziness*: The backbone of slags, as well as the elements attached to this backbone, are handled lazily — i.e., they are only computed if and when they are needed. Laziness helps to realize operational faithfulness by controlling the order of operations and guaranteeing that no spurious operations are performed within a sliver network.

Slags transmit intermediate values and synchronization information between concurrently executing slivers. Each slag element represents the information produced or consumed by one recursive layer of a sliver. Slags are realized as lazy data structures that carry a pair of call/return synchrons for every lazily-held element. Slivers are realized as procedures that consume and/or produce slags and also follow an important set of conventions for manipulating the elements and sychronizing on the synchrons. The conventions guarantee that sliver processing proceeds in lock step and that undesirable intermediate storage does not accrue.

1.3 Alternate Perspectives on This Research

The main theme of this research is that it is possible to design modular programs without necessarily sacrificing control. However, there are some alternate perspectives that characterize this work or portions thereof:

1. *Operational Modularity*: Traditional black-box abstraction techniques exhibit *functional modularity* in the sense that they define the functional input/output behavior of a modular component. This work explores the notion of *operational modularity* — decomposing the complex operational behavior of a monolithic system into simpler

parts. Sliver decomposition achieves a kind of operational modularity by widening the interface of the traditional aggregate data approach to include operationally significant synchronization information.

2. *A Dynamic Model of Lock Step Processing*: Numerous program transformations and compilation techniques exist for removing intermediate data structures from aggregate data programs [DR76, Dar82, Bac78, Bel86, Bir89a, Bir86, Bir88, GW78, Bud88, Wad84, Wad85, Wad88, Chi92, GLJ93, Wat91]. Most of these compile-time techniques are based on a high-level version of the *loop fusion* technique employed by many optimizing compilers [ASU86]. Synchronized lazy aggregates are essentially a mechanism for performing loop fusion at run-time. Due to their dynamic nature, synchronized lazy aggregates enable a level of expressiveness that cannot be matched by the static approaches.

3. *Hybrid SPS Techniques*: Sliver decomposition can be viewed as an answer to the riddle "What do you get if you cross aggregate data with channels?" That is, it is an attempt to combine the best aspects of a number of existing SPS techniques into a single technique. Sliver decomposition extends lazy aggregates with the synchronization of channel-based approaches.

4. *Generalizing Series*: Much of this research was inspired by Waters's extensive work on loop decomposition [Wat78, Wat79, Wat84, Wat87, Wat90, Wat91]. Waters designed a well-engineered mechanism for expressing loops in terms of networks of linear iterative operators manipulating a kind of synchronized lazy data structure known as *series*. In addition, he developed conditions and static analysis techniques guaranteeing that a large class of series networks can be efficiently compiled into loops. These make programs expressed in terms of series a practical alternative to loops.

   The research described here is a first step towards extending series to handle general linear recursion (not just loops) and general tree-shaped computations. Slags are a generalization of series that support these more complex computational shapes. But currently, slags are only explained in terms of a somewhat inefficient dynamic synchronization model. In order to make sliver decomposition a practical alternative to monolithic recursions, it will be necessary to develop series-like static analysis and compilation techniques for sliver programs.

5. *Abstracting Over Hughes's Ideas*: In his dissertation [Hug83] and an important but little-known paper [Hug84][2] , Hughes explains why concurrency and synchronization are necessary for preserving the space characteristics of a monolithic program in a modular SPS program. He introduces concurrency and synchronization constructs that can be thought of as annotations for controlling the operational behavior of a functional program. The concurrency and synchronization techniques used in sliver decomposition are similar to ones introduced by Hughes, but they are organized into abstractions that make them easier to program with and reason about.

6. *First-class Synchronization Barriers*: The Id programming language employs a *synchronization barrier* construct as a means of controlling the non-functional features of a mostly functional language [Bar92]. The synchrons introduced in this report can be viewed as first-class synchronization barriers. Thus, one aspect of this research is exploring the gains in expressive power that can be achieved by making synchronization barriers first-class objects.

7. *A Pedagogically Viable Graphical Programming Model*: In part, sliver decomposition was motivated by a desire to develop a graphical evaluation model for the Scheme programming language. Here, "graphical" is meant in two senses: (1) "graph-theoretic" – i.e., is expressed in terms of vertices and edges; and (2) "visual" – i.e., takes advantage of human visual perception skills. The goal was to develop a graphical model that:

   (a) Explains all the important features of Scheme (e.g., first-class procedures, tail-recursion, side-effects, continuations).

   (b) Is straightforward enough to be automatically animated.

   (c) Is simple enough to be understood by novice Scheme programmers.

   (d) Is an effective pedagogical tool for teaching Scheme.

[2] I am indebted to Ignacio Trejos-Zelaya for tracking down this paper at Oxford and sending me a copy.

The EDGAR model introduced in this dissertation satisfies the first two criteria; the last two need to be empirically verified. A prototype implementation of an EDGAR-based graphical programming model has been implemented and shows promise as a pedagogical tool. An unexpected benefit of this research is that the model is able to explain important concurrency features that are not a part of standard Scheme.

1.4 Dissertation Road Map

The remaining text of this dissertation is organized into three major segments:

I. BACKGROUND

The background segment gives a detailed motivation of the problem. Readers who are eager to learn about the details of the sliver technique should skip ahead to the next segment. There are two chapters in the background segment:

- **Chapter 2: Slivers** – A motivation for sliver decomposition in the context of two monolithic programs: an employee database program and an alpha renaming program.
- **Chapter 3: The Signal Processing Style of Programming** – A detailed analysis of why existing SPS techniques fail to express desirable operational characteristics of programs.

II. SLIVER DECOMPOSITION

The sliver decomposition segment gives a detailed exposition of slivers and slags. These are complex entities involving numerous subtleties. In order to suppress a flood of potentially overwhelming details, they are presented in a top-down fashion over five chapters:

- **Chapter 4: Computational Shape** – A presentation of a simple notion of computational shape. Shapes are described in terms of the time-based ordering induced on the call and return events in the execution of a recursive procedure.

- **Chapter 5: Synchronized Lazy Aggregates** – An explanation of the lock step processing model underlying the sliver technique. Synchronized lazy aggregates are introduced as a mechanism for guaranteeing that networks of slivers simulate the behavior of a corresponding monolithic procedure.

- **Chapter 6:** SYNAPSE**: Programming with Slivers and Slags** – An illustration of the power of slivers and slags in the context of SYNAPSE, a simple language for manipulating synchronized lists and trees.

- **Chapter 7:** OPERA**: Controlling Operational Behavior** – A presentation of OPERA, the concurrent dialect of Scheme in which SYNAPSE is embedded. An informal description of OPERA's concurrency, synchronization, and non-strictness features is followed by an explanation of how SYNAPSE is implemented in OPERA.

- **Chapter 8:** EDGAR**: Explicit Demand Graph Reduction** – An overview of EDGAR, an explicit demand graph reduction model that provides an operational semantics for OPERA. OPERA's concurrency, synchronization, and non-strictness mechanisms are formally described here.

The top-down approach effectively manages complexity, but suffers a major drawback: the discussion of many concepts is distributed across several chapters. For instance, crucial notions like demand-driven evaluation, concurrency, synchronization, non-strictness, and tail-recursion are first introduced in an informal, almost hand-waving, fashion; then their details are unravelled over several chapters. Readers who prefer bottom-up presentations are encouraged to jump ahead to the detailed expositions in the later chapters. Skimming the SYNAPSE programs in Chapter 6 and the graphical rewrite rules in Chapter 8 may be particularly helpful for building intuitions.

III. WRAP-UP

The main text of the dissertation concludes with a wrap-up segment of two chapters and an appendix:

- **Chapter 9: Experience** – A discussion of the experimental aspects of the research, including the implementation and testing of EDGAR, OPERA, and

SYNAPSE. This chapter also describes the DYNAMATOR, a graphical program animator that proved invaluable in the development of the other systems.

- **Chapter 10: Conclusion** – A summary of the research, including contributions and future work.
- **Appendix A: Glossary** – This dissertation introduces a large number of new terms, and uses some existing terms in a non-standard way. The glossary is provided to help the reader adjust to the terminology.