Slivers:
Computational Modularity via Synchronized Lazy Aggregates

by

Franklyn Albin Turbak

S.B., Massachusetts Institute of Technology (1986)
S.M., Massachusetts Institute of Technology (1986)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1994

Signature of Author ....
Department of Electrical Engineering and Computer Science
January 31, 1994

Certified by ....
Gerald Jay Sussman
Matsushita Professor of Electrical Engineering
Thesis Supervisor

Certified by ....
David K. Gifford
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Slivers:
Computational Modularity via
Synchronized Lazy Aggregates

by

Franklyn Albin Turbak

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 1994, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

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.

Thesis Supervisor: Gerald Jay Sussman
Title: Matsushita Professor of Electrical Engineering

Thesis Supervisor: David K. Gifford
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

Many people skip over the acknowledgments when they read a book. Not me. Every book has a hidden story that isn't told by the main text. The acknowledgments give a glimpse into the *process* by which the book was created — how the ideas took shape, who was involved when, the emotional ups and downs of the author, and so on. In fact, you might say that the process of writing a thesis has a shape — but that's another thesis.

What follows is a glimpse into the story of this thesis. First, my thesis committee:

Gerry Sussman (thesis co-supervisor) and Hal Abelson (thesis reader) are ultimately responsible for this work. They created The Course (6.001, MIT's introductory computer science course) and wrote The Book (*Structure and Interpretation of Computer Programs*) that changed my life. I wasn't planning to major in computer science as an undergraduate, but after their course, I couldn't imagine doing anything else. A religious experience? You might say that.

Their book said some crazy things about computations having shape. This planted seeds in my head that germinated years later. I decided to try to make sense out of this computational shape notion. This thesis represents a checkpoint in that process. There's still a long way to go.

Gerry is the archetypical hacker, mastering everything from watch repair to solar system dynamics. His unbounded energy, infectious enthusiasm, diverse interests, and good-natured spirit recharged me again and again during the long haul of this research. Hal is simply the finest teacher and technical writer I have ever known.

Together, Gerry and Hal are the Lennon and McCartney of Computer Science. They would probably hate this title, since they'd like to be associated as little as possible with computer science. (I'm not sure about their feelings on the Beatles.) After all, in 6.001, don't they teach that computer science is not a science, and has very little to do with computers? But like it or not, it's true. They have put out more high quality teaching and research than anyone else I have ever seen. They are the kind of pair that inspire legends, and about whom ballads are written. Everything they touch, they improve.

I learned from them that you really don't understand something unless you can boil it down into a 6.001 problem set. I hope to spend a large chunk of the rest of my life boiling things down in this manner.

David Gifford (thesis co-supervisor) got me hooked on semantics. He developed The Other Course in my life (6.821, MIT's graduate programming languages course). I have learned an immense amount about programming languages and systems while under Dave's tutelage. Over the years, Dave has provided me with lots of support, encouraged me to formalize my fuzzy ideas, and steered my thinking in more practical directions.

Dick Waters (thesis reader) has helped me more on particulars than anyone else. He has spent many Thursday afternoons talking with me about both high level issues and nitty gritty details. After many years of struggling to explain my ideas to others, it was exciting and refreshing to talk shop with Dick. My only regret is that I added him to my thesis committee at such a late stage!

David McAllester (thesis reader) engaged me in stimulating discussion, and was the source of many neat ideas.

Now onto my family and friends:

First and most important is my best friend, the Love Of My Life, and my wife: Lisa Savini. I don't know why, but for some reason wives almost always get relegated to the last line of the acknowledgments. Lisa deserves better than that. Again and again, her love has lifted me out the the depths of despair, her conversation has kept me sane, and her tasty cooking has nourished me. Lisa gradually assumed all household duties while her husband mutated into a zombified hermit. And in the homestretch, she proofread the entire document. I look forward to getting to know her again!

I wouldn't have made it to the brink of doctorhood without the support of my family. Mom and Dad raised me in an intellectually stimulating environment and gave me more love and encouragement than any child deserves. They have been waiting for this document for a long time! I am honored to join my father as another Dr. Turbak.

My brother, Stephen, my sister-in-law, Michelle, and my two nephews, Casimir, and Darius, have strived to keep me in touch with reality during my long thesis journey. I am ever so grateful that I finished my doctorate before Caz joined me here at MIT (he is now almost five years of age).

My new family, the Savinis, provided me with large quantities of love and food (is there a distinction?) throughout the past few years.

Jonathan Rees is one of my heroes. He has the uncanny ability to give crystal clear explanations of most any topic in real time. I've learned more about programming language design and good programming style from Jonathan than from any other source. When my interest in my thesis waned, Jonathan convinced me that I was working on something worthwhile. He also suggested many improvements to the organization of my dissertation. His working in the office next door to mine during the final stages of my dissertation was a godsend.

Brian Reistad became a Good Friend who provided detailed feedback about the document, was always willing to listen about details, and checked up on me daily when I was entombed in my office.

David Espinosa showed up in my life at just the right time. He renewed my excitement in programming languages just when my enthusiasm was starting to flag. He also gave me lots of feedback about my research and this document.

Jim O'Toole suggested many valuable improvements for restructuring the presentation of my work.

Mark Sheldon (a.k.a. Eldo) helped to keep me afloat with his continually bubbly demeanor and his conversation, both technical and non-technical.

Alan Bawden introduced me to the nuances of graph reduction and taught me lots of Cool Things.

Feng Zhao swapped thesis ideas with me on a weekly basis during the early stages of my research. I am grateful for his friendship, and for being a sounding board for all my fuzzy thoughts.

Mark Day shared my original vision about capturing the space/time behavior of processes, and has provided valuable comments and suggests along the way.

Ziggy Blair was one of the few people who voiced appreciation for my research in the early stages when most everyone else was giving me icy stares.

Bill Rozas tutored me in a wide range of computer science topics in the process of answering gazillions of my questions.

The Switzerland crew — Hal and Gerry's group — are an amazing collection of incredibly smart and helpful people with refreshing views on just about any topic you can imagine. In addition to the folks listed above, Stephen Adams, Andy Berlin, Mark Friedman, Arthur Gleckler, Philip Greenspun, Chris Hanson, Elmer Hung, Brian LaMacchia, Jim Miller, Ognan Nastov, Jacob Katzenelson, Kleanthes Koniaris, Nick Papadakis, Thanos Siapas, Pete Skordos, Rajeev Surati, and Henry Wu all help to make the fourth floor of Tech Square a very exciting environment.

The Swiss graduates – Liz Bradley, Mike Eisenberg, Mitch Resnick, Ken Yip, and Feng Zhao – awed and inspired me while they were here, and were good friends to boot. I miss them all dearly.

For their feedback and encouragement on my thesis research, I am grateful to Andy diSessa, Ian Horswill, Trevor Jim, Pierre Jouvelot, Jintae Lee, Nate Osgood, John Pezaris, Roberto Segala, Ellen Spertus, and Julie Sussman.

Becky Bisbee, Jeanne Darling, and Marilyn Pierce helped me out by taking care of lots of details pertaining to my thesis and my life as a graduate student.

I am indebted to Ignacio Trejos-Zelaya, who was able to track down Hughes's "Parallel Functional Languages Use Less Space" in a forsaken file cabinet at Oxford when nobody else in the world could seem to find a copy.

I am grateful to all the friends and family who refuse to give up on me yet even though I've totally neglected them for a long time now. Special thanks to Douglas Massidda and Fatima Serra for sharing the bounty of the ocean with Lisa and me; to David Chiang for recharging our friendship every time he comes to Boston; to Robert Kwon for calling me up from Japan every once in awhile; to Chablo Boyadjis, Mike Dawson, Nick Newell, and Jean Spence for some great hiking trips; to Christine Allan, Dan Frost, and Tina Katkocin for checking up on me; to Andy Litman, Debbie Utley, Ken & Ginny Grant, and Linda & Nitin Upadhyaya for sharing their homes with Lisa and me; and especially to the Paulist Center's Wednesday Night Supper Club for helping me find Lisa.

Finally, I would also like to acknowledge the inventor of acknowledgments, without whom this section would not have been possible.

This report 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.

Note to the Reader

This dissertation will be revised and published as MIT Artificial Intelligence Laboratory technical report AI-TR-1466. Readers are encouraged to consult the technical report for various extensions to, and simplifications of, the work described here.

Contents

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`),

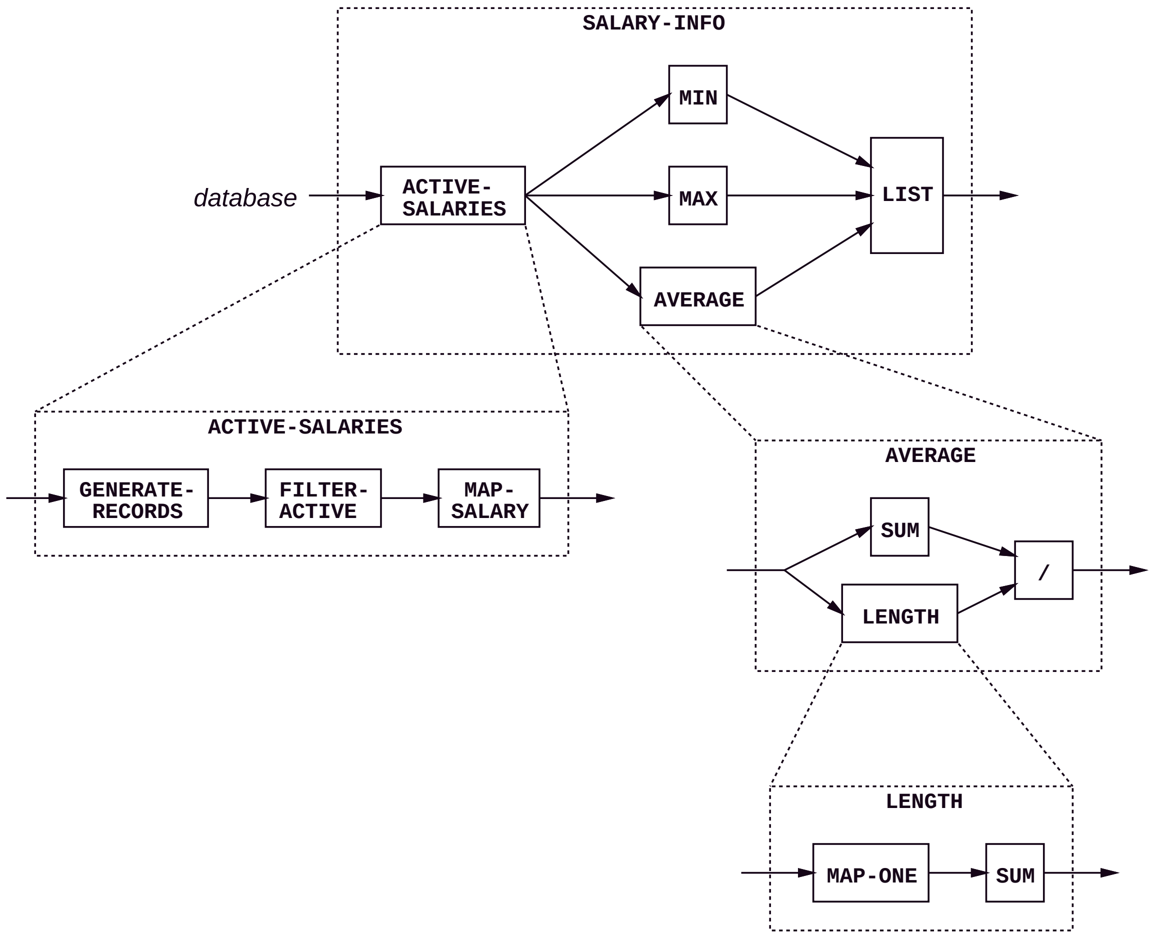


Figure 1.1: Structure of a modular program for computing the minimum, maximum, and average salaries for active employees from a given database.

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,

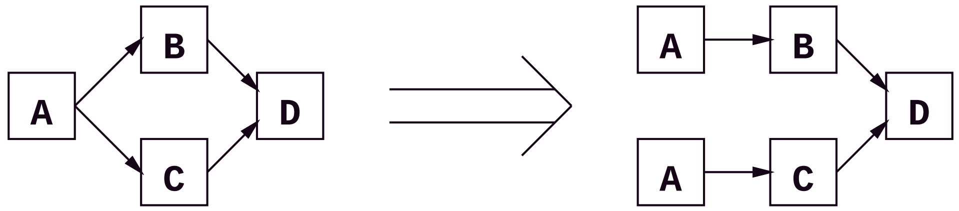


Figure 1.2: In SPS techniques that do not allow fan-out, a device must be replicated.

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.)

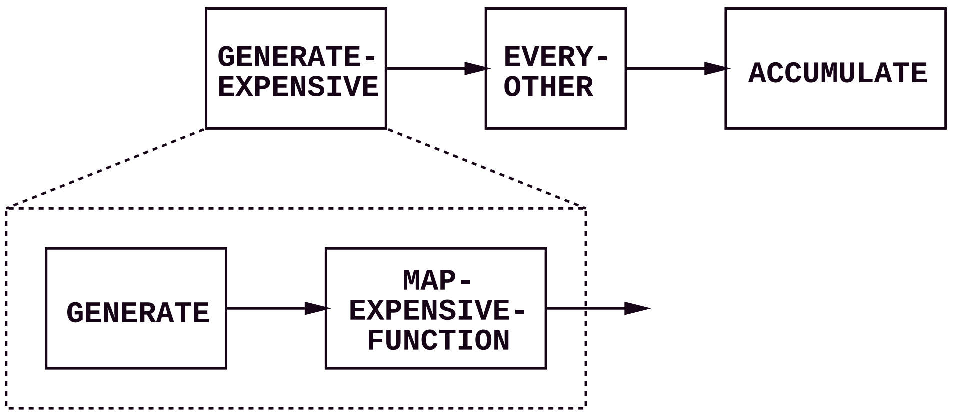


Figure 1.3: A network for which many SPS techniques perform unnecessary work.

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,

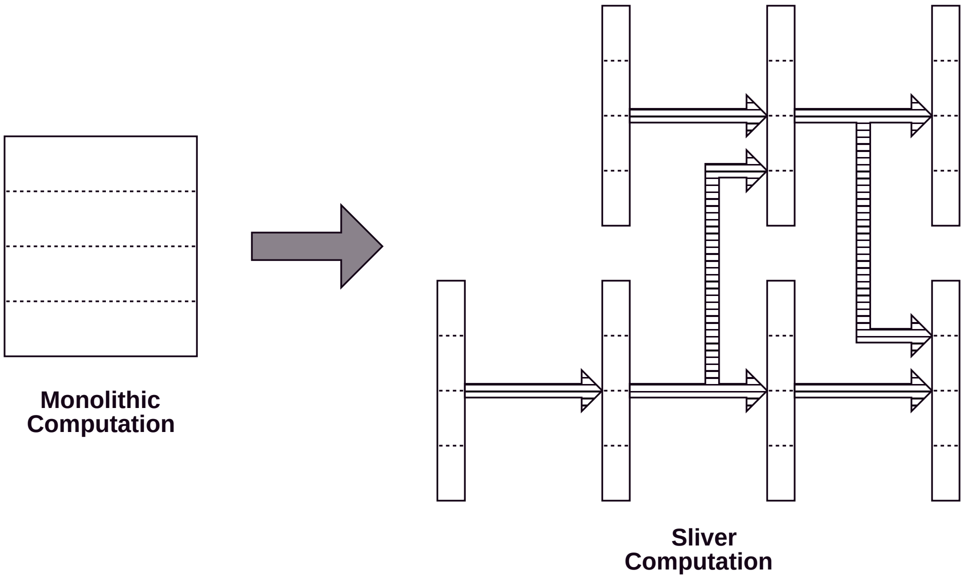


Figure 1.4: Decomposing a monolithic computation into a network of slivers.

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-

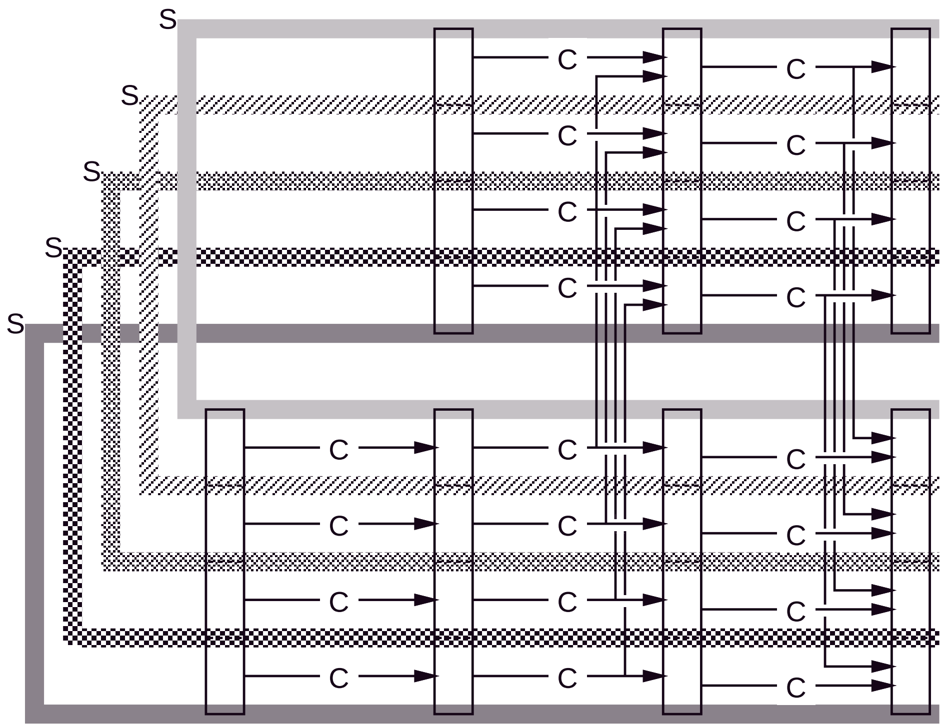
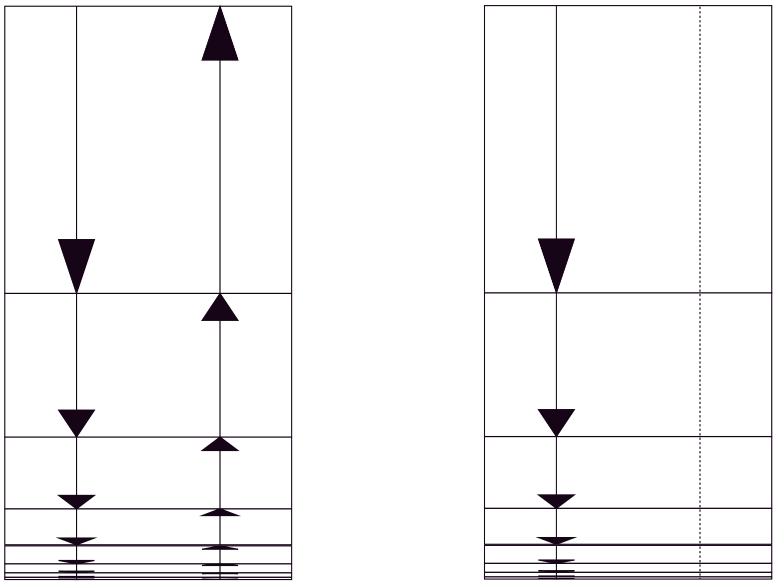


Figure 1.5: Communication events (C) and synchronization events (S) among the slivers in a network.

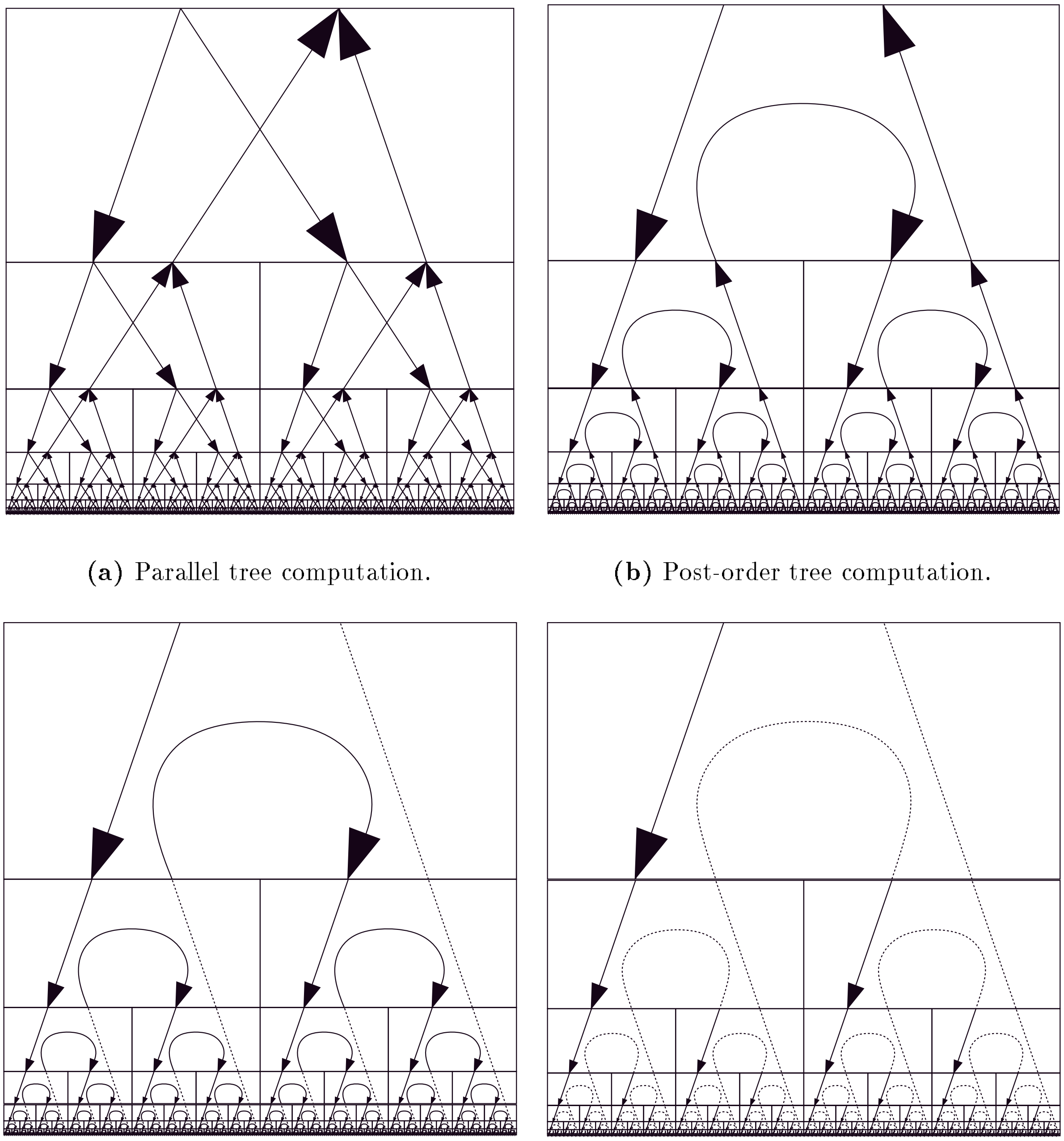


(a) Shape of a general linear recursion. **(b)** Shape of a linear iteration.

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.

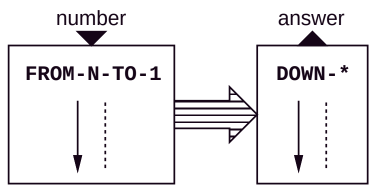
Sliver decomposition is intended not to replace other SPS techniques, but to be used in conjunction with them. The lock step processing of sliver network is not appropriate for many computations. However, sliver decomposition interfaces nicely with other aggregate data mechanisms, so it is easy to flexibly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

1.2.2 Some Simple Examples

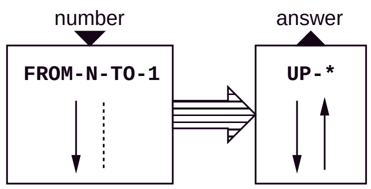
In this section, I present a few simple examples that give the flavor of sliver decomposition and hint at its expressive power. (The examples in this section are necessarily brief and simple. The reader is encouraged to explore the more interesting examples in Chapter 6.)

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.

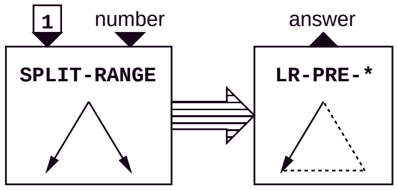
In **(a)**, the `FROM-N-TO-1` sliver generates the integers from the input down to (and including) 1, while the `DOWN-*` iteratively accumulates these numbers. The downward arrows annotating the slivers and the "`DOWN`" in `DOWN-*` indicate that both slivers have only a down phase, so the resulting computation 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



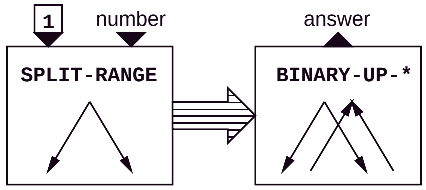
(a) Linear iterative factorial.



(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

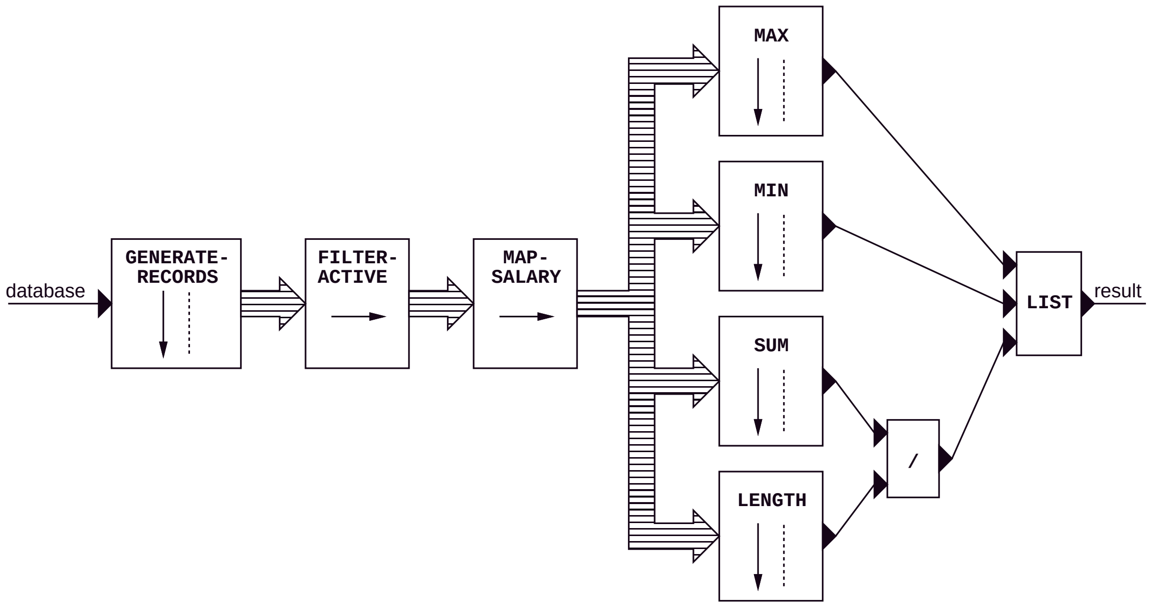


Figure 1.9: The salary information program expressed as a sliver diagram. The thick cables represent slags, while the thin lines represent non-slag data. The whole network behaves like a monolithic iteration because each of the components is inherently iterative.

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.

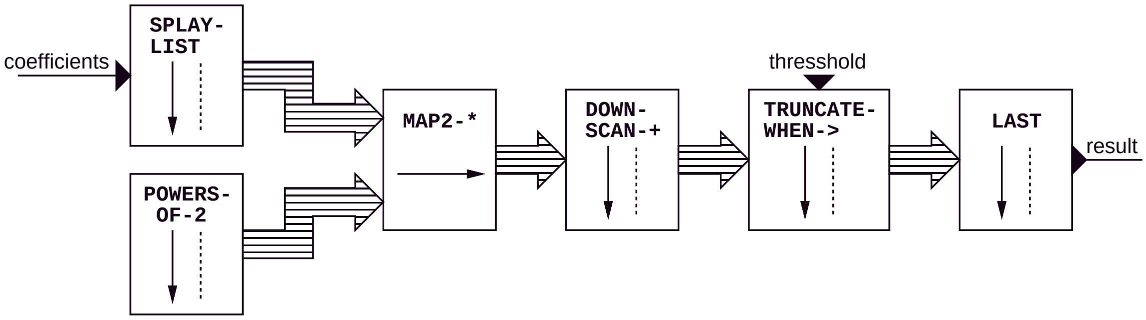


Figure 1.10: A sliver diagram introducing some new kinds of linear slivers.

- `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*.

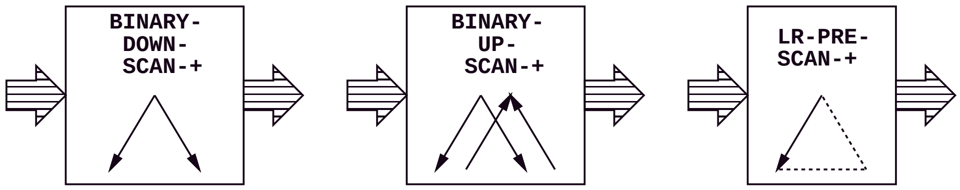


Figure 1.11: Three tree scanners distinquished by shape.

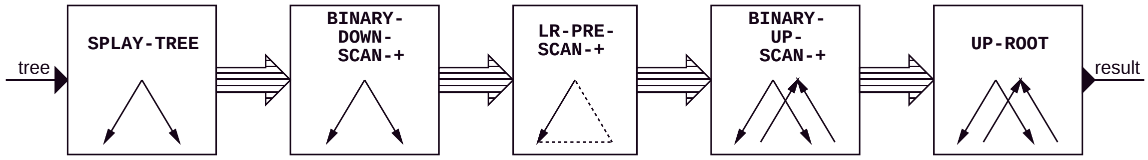


Figure 1.12: A sequential composition of differently shaped tree scanners.

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

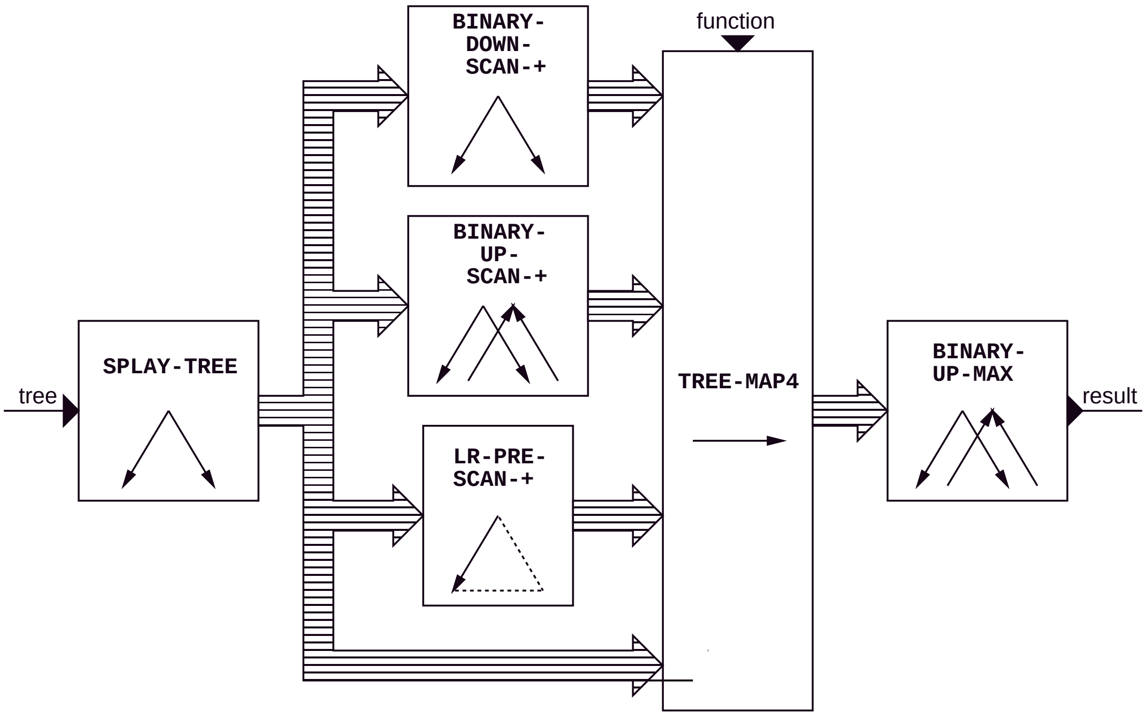


Figure 1.13: A parallel composition of differently shaped tree scanners.

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.

Chapter 2

Slivers

This chapter introduces a class of programming idioms that I call *slivers*. Slivers capture the generating, mapping, filtering, and accumulating idioms commonly found in loops and recursive procedures. Many programs can be visualized as *sliver diagrams* in which information flows through a static network of slivers. This type of program organization is certainly not new, and in the next chapter we will see how it is supported in various popular programming paradigms. What *is* new is a dynamic framework for combining the operational aspects of slivers in a reasonable way. I will only hint at that framework in this chapter, but will develop it in detail in Chapters 4 and 5.

I motivate slivers in the context of two extended examples:

1. Simple manipulations of a linearly-structured database. This example introduces many of the issues relevant to modularity in the signal processing style of programming.

2. Alpha renaming of lambda calculus terms. This tree-structured example illustrates important issues and patterns of computation that do not arise in linear examples.

For the purpose of presentation, both examples have intentionally been kept simple. However, even though the examples are somewhat contrived, the issues they raise are very real.

2.1 Linear Example: Database Manipulation

2.1.1 Overview

In this example, we consider simple programs manipulating an employee database. Suppose that the interface to the database is the following set of procedures:[1]

- `(first-record` *database-descriptor*`)` returns the first record of the specified database.
- `(next-record` *record*`)` returns the database record following the given one.
- `(end-of-database?` *record*`)` tests for a distinguished database termination record.
- `(record-get` *record field*`)` retrieves the contents of the specified field from the given record.

The first three procedures effectively make the database accessible as a linked list of records. The `record-get` procedure is the means of extracting information from an individual record. For the purposes of this example, we will assume that the records in every database are sorted alphabetically by surname.

The procedural interface hides many details about how the database is implemented. For all we know, the database records might be stored in a tree-like fashion; different fields might be stored in distinct tables; or parts of the database may be stored remotely, perhaps even distributed across several physical servers and sites. Even a bizarre scenario in which every call to `next-record` initiates a request for a data entry operator to type in the next employee record on the fly is consistent with this interface! These details do not affect the *values* returned by the procedures, although they may show through in other ways (e.g., `next-record` may take a long time if records are stored remotely).

Below, we investigate two procedures describing computations on an employee database:

- `(mean-age` *database-descriptor*`)` returns the average age of employees in the specified database.

[1] The procedure specifications have parentheses because they, like all programming examples in this dissertation, are written in Scheme, a dialect of Lisp ([CR+91], [ASS85]). I use Scheme because its support for first-class procedures, side effects, and tail recursion permits concise expression of a wide range of programming styles. However, the computational issues I am investigating are independent of the particular language in which the examples are phrased.

- (fat-cats *threshold database-descriptor*) returns a list (sorted alphabetically) of the names of all employees in the specified database whose salaries are greater than the given threshold.

These procedures will serve as a basis for comparing several styles of programming. We will explore issues of modularity by considering the ease with which these procedures can be modified and combined.

2.1.2 Monolithic Style: Functional Paradigm

In the *monolithic style*, the example programs are implemented as single recursive procedures that collect information during a traversal of the database. Here is a monolithic implementation of mean-age written in the functional programming paradigm:[2]

```
(define (mean-age_fun database)
  (define (loop record age-total count)
    (if (end-of-database? record)
        (/ age-total count)
        (loop (next-record record)
              (+ age-total (record-get record 'age))
              (+ 1 count))))
  (loop (first-record database) 0 0))
```

The internal loop procedure performs an iterative traversal of the database while maintaining three state variables: record points to the current record, age-total names the running sum of employee ages, and count names the number of records examined.

The monolithic functional version of fat-cats has a similar structure:

```
(define (fat-cats_fun threshold database)
  (define (gather record)
    (if (end-of-database? record)
        '()
        (if (> (record-get record 'salary)
               threshold)
            (cons (record-get record 'name)
                  (gather (next-record record)))
            (gather (next-record record)))))
  (gather (first-record database)))
```

[2]Identifiers naming procedures (such as mean-age) will often be subscripted (in this case, with $_{fun}$) to distinguish different implementations of the same procedure. The subscript is *not* part of the identifier; it is merely a convenient way to refer to a particular definition.

The internal `gather` procedure traverses the entire database and collects into a list the names of the employees satisfying the salary predicate. The list collection strategy used here, in which an employee name is prepended to the list resulting from a recursive call to `gather`, preserves the relative ordering of the selected employees. Because the databases are ordered alphabetically, so is the resulting list.

2.1.3 Monolithic Style: Imperative Paradigm

Both of the above procedures are written in the functional paradigm, which does not permit assignment. For comparison, Figure 2.1 presents monolithic versions of the two procedures written in the imperative paradigm. The imperative programs are rather similar to their functional counterparts. The main difference is that immutable formal parameters in the functional versions become mutable state variables in the imperative version.

2.1.4 Computational Shapes

Following [ASS85], I carefully distinguish *procedures* from the *computations* that they specify.[3] A procedure is just a specification for a computational process, whereas a computation is the process that dynamically unfolds when the procedure is called.

Despite the fact that the functional and imperative versions of `mean-age` are written in different styles, they specify computations with similar operational characteristics. In both cases, even though the internal `loop` procedure is syntactically recursive, the tail-recursive property of Scheme ([CR+91], [Ste77]) guarantees that `loop` executes iteratively, just as if it had been written as a `do`, `for`, or `while` loop in other languages. And both `loop` procedures iterate over the same three state variables, though in one case they are explicit arguments and in the other case they are implicit. The operational behavior of $\texttt{fat-cats}_{fun}$ and $\texttt{fat-cats}_{imp}$ is likewise very similar.

On the other hand, even though $\texttt{mean-age}_{fun}$ and $\texttt{fat-cats}_{fun}$ are both written in the functional style, they specify computations that differ in fundamental ways. In addition to the obvious fact that these programs perform different calculations, the `gather` procedure

[3] I use the term "computation" in place of the term "process" used by [ASS85]. I make this change to distinguish the standard usage of "process" in the concurrency community from the notion for computational unfoldment presented in [ASS85].

```
(define (mean-age_imp database)
  (let ((record (first-record database))
        (age-total 0)
        (count 0))
    (define (loop)
      (if (end-of-database? record)
          (/ age-total count)
          (begin
            (set! age-total (+ age-total (record-get record 'age)))
            (set! count (+ count 1))
            (set! record (next-record record))
            (loop))))
    (loop)))

(define (fat-cats_imp threshold database)
  (let ((record (first-record database)))
    (define (gather)
      (if (end-of-database? record)
          '()
          (if (> (record-get record 'salary)
                 threshold)
              (let ((name (record-get record 'name)))
                (begin
                  (set! record (next-record record))
                  (cons name (gather))))
              (begin
                (set! record (next-record record))
                (gather)))))
    (gather)))
```

Figure 2.1: Imperative versions of the database procedures.

does not have the purely iterative character of `loop`. When the salary predicate is satisfied, the implementation must "remember" to perform a pending `cons` operation upon returning from the recursive call to `gather`. Conceptually, such calls to `gather` push information on an implicit control stack. Calls made to `gather` when the salary predicate is not satisfied push no information on the stack (they act like `goto`s). The amount of control space required to execute `fat-cats` is thus proportional to the number of names satisfying the salary predicate.

We'd like some method of characterizing these sorts of operational similarities and differences between computations. One way of doing this is to adopt a standard evaluation model that captures operational features. For example, [ASS85] uses an expresssion-rewriting model to analyze procedures in terms of traces of their evaluation. Here is a summary of such a trace of $\texttt{mean-age}_{fun}$ on a sample database (where the details of database and record representations have been surpressed):

```
(mean-age_fun  database)
(loop  record1    0 0)
(loop  record2   43 1)
(loop  record3  103 2)
(loop  record4  125 3)
(loop  record5  174 4)
(loop  record6  202 5)
(loop  record7  233 6)
...
```

On every line of the trace, the state of the computation is entirely captured in the values of the three arguments to `loop`. Thus, $\texttt{mean-age}_{fun}$ exhibits the constant space behavior of an iterative computation.

In contrast, here is a trace for `fat-cats` on a sample database:

```
(fat-cats_fun 250000  database)
(gather  record1)
(gather  record2)
(cons "Raws P. Arrow" (gather  record3))
(cons "Raws P. Arrow" (gather  record4))
(cons "Raws P. Arrow" (gather  record5))
(cons "Raws P. Arrow" (cons "Gill Bates" (gather  record6)))
(cons "Raws P. Arrow" (cons "Gill Bates" (gather  record7)))
...
```

Some calls to `gather` simply rewrite to another call of `gather` on the next record. However, when the salary predicate is satisfied, a call to `gather` rewrites to a `cons` application whose second argument is a call to `gather`. The rightward-growing "bulge" of the trace expressions is due to the pending calls to `cons`, which textually encode the implicit control stack required by the computation.

These examples show how the expression-rewriting model differentiates procedures according to the pattern by which their computations evolve. In keeping with [ASS85], I will refer to these patterns of computational evolution as *shapes of computation*. The notion of shape is intended to capture operational features of a computation, such as time and space complexity and the relative ordering of various events. The work described in this report was motivated by the desire to formalize the notion of computational shape and to incorporate shape-based ideas into a programming language.

Certain patterns are so common that programmers give them names. For example, `mean-age` is a *linear iteration* in three state variables while `fat-cats` is a *linear recursion* in one argument. Both shapes are said to be linear because because each line of the trace contains only a single recursive call. Later we will study more general *tree shapes* in which several recursive calls are potentially active.

2.1.5 Monolithic Programs Lack Modularity

`Mean-age` and `fat-cats` exhibit several common idioms for linear computations. Despite the differences in their shape and what they compute, both procedures generate all the records of the database in succession, and accumulate information from each record. `Mean-age` uses two instances of a running sum idiom, in which a numeric variable initialized to 0 is used to accumulate a sum. `Fat-cats` uses a filtering idiom to eliminate unwanted records, a mapping idiom to find the name in each record, and a list accumulation idiom to collect information in a list. These kinds of idioms (generate, map, filter, accumulate) arise repeatedly in manipulation of linear data. For example, Waters found that 90% of the code in the Fortran Scientific Subroutine Package could be expressed wholly in terms of these idioms [Wat79].

A major problem with the above procedures is that the idioms are not localized in the program text. For example, the database enumeration idiom is spread out across each procedure body in the calls to `first-record`, `next-record`, and `end-of-database?`. Similarly, the running sum idiom conceptually consists of the declaration, initialization, and update of a variable that maintains the sum. However, in `mean-age`$_{fun}$, these three parts of the running sum idiom are textually separated from each other. The situation is marginally better in `mean-age`$_{imp}$, where the declaration and initialization occur together.

The problem with non-localized idioms is that they are hard to read, write, and modify. Idiom recognition is hampered when the programmer has to hunt for fragments of an idiom in different parts of the code. First, it is necessary to make sure that all the right pieces are present, and then it is necessary to become convinced that the rest of the code doesn't prevent the idiom from working as expected. Reasoning is similarly complicated when writing or modifying code; the programmer has to mentally juggle pieces from various idioms and guarantee that they don't adversely interact. The non-locality forces the programmer to wade through details unrelated to the idiom.

With non-localized idioms, the author of a programs may very well be guided by mental notions of such idioms, and an attuned reader of programs can recognize the idioms. But the idioms are only *implicit*. A fundamental principle of modularity in programming is that idioms should be made *explicit*. When idioms are captured and named, programs can be expressed by explicitly composing program fragments embodying the idioms. The above procedures are said to be *monolithic* because they are expressed as densely interwoven idiom fragments rather than as compositions of reusable idiom components.

Programs written in the monolithic style are difficult to combine and modify. For example, suppose we want to compute the average age of employees earning more than a given salary. Intuitively, we would like to connect `fat-cats` and `mean-age` in series. But the result of `fat-cats` is a list of names, and `mean-age` expects a database. Even if we reach inside the procedures, we do not find a common interface through which they can be connected. Instead, the monolithic style forces us to construct an entirely new procedure from scratch:[4]

[4] Here, and throughout the rest of this section, we will only consider procedures written in the functional style. However, all the analyses and conclusions hold for the imperative style as well.

```
(define (fat-cat-age_fun threshold database)
  (define (loop record age-total count)
    (if (end-of-database? record)
        (/ age-total count)
        (if (> (record-get record 'salary)
               threshold)
            (loop (next-record record)
                  (+ age-total (record-get record 'age))
                  (+ 1 count))
            (loop (next-record record)
                  age-total
                  count))))
  (loop (first-record database) 0 0))
```

Almost all of the expression fragments used in the resulting procedure are taken from the two original functional procedures. But we cannot point to single entities that represent the salary filtering, the running total for ages, or the running count of filtered employees. As before, these idioms are smeared throughout the (new) `loop` procedure. The work that went into implementing these idioms in the original procedures must be repeated because the monolithic style does not create reusable language artifacts embodying the idioms.

An interesting feature of `fat-cat-age`$_{fun}$ is that it is iterative, even though it is partially derived the non-iterative `fat-cats`$_{fun}$ procedure. Examination of that procedure reveals that the stacking behavior is due only to the list accumulation idiom. Since `fat-cat-age`$_{fun}$ does not use that idiom, it does not require a procedure call stack. This example suggests that we should be able to associate shapes of computation with individual idioms and then determine the shape generated by a program from the shapes of the idioms out of which it is constructed. Later, we will explore this idea in depth.

As another example, consider a procedure that returns both the fat cats and the average employee age for a given database. Here we want to connect `mean-age` and `fat-cats` in parallel. In the simplest approach, we can just encapsulate the two existing computations within one procedure:

```
(define (fat-cats&mean-age_nice threshold database)
  (list (fat-cats threshold database)
        (mean-age database)))
```

This extremely simple means of combination is a key advantage of the black-box modularity offered by procedures. This procedure works regardless of the styles in which `fat-cats` and `mean-age` happen to have been written.

Unfortunately, while this procedure computes the desired *values*, it doesn't necessarily compute them in the desired *way*. In particular, the calls to `fat-cats` and `mean-age` will each traverse the entire database independently. This means that `next-record` will be called twice for every record in the database — once by `fat-cats` and once by `mean-age`. The overhead of calling `next-record` twice for every record may be deemed unacceptable, especially in the case where calls to `next-record` are particularly expensive (e.g., when the database is stored remotely).

An alternative is to merge the computations of `fat-cats` and `mean-age` such that the database is traversed only once. Here is the merged version for the functional implementations:[5]

```
(define (fat-cats&mean-age_fun threshold database)
  (define (both record age-total count)
    (if (end-of-database? record)
        (list '() (/ age-total count))
        (if (> (record-get record 'salary)
               threshold)
            (mlet (((rest-names avg)
                    (both (next-record record)
                          (+ age-total (record-get record 'age))
                          (+ 1 count))))
              (list (cons (record-get record 'name) rest-names) avg))
            (both (next-record record)
                  (+ age-total (record-get record 'age))
                  (+ 1 count)))))
  (both (first-record database) 0 0))
```

Here, the internal `both` procedure performs a single traversal of the database, during which it accumulates both a list of names and an average age. It returns the two results as a two-element list. Since this procedure does use the linearly-recursive list accumulation idiom, its shape is a linear recursion rather than a linear iteration.

Although `fat-cats&mean-age`$_{fun}$ is "better" than `fat-cats&mean-age`$_{nice}$ in terms of execution time, it is a whole lot worse in terms of understandability and modifiability. The idioms here are exactly the same as those used in `fat-cat-age`, but their interleaving makes

[5] I assume that Scheme has been extended with a pattern matching version of `let` called `mlet`. A `mlet` expression has the same form as a `let` expression, except that the name position of a binding may contain a *name pattern* rather than just a single name. A name pattern is any list structure whose atomic elements are names. The name pattern is matched against the value of the binding expression, and the resulting name/value associations are accessible in the body of the `mlet`. It is an error if a name pattern does not match the binding value.

It is possible to dispense with `mlet` by instead using explicit selectors or Scheme's multiple value return mechanism. However, I find both of these alternatives too unwieldy for expository purposes.

them even harder to recognize. Additionally, the program is complicated by the details of managing two return values rather than one. For example, the fact that the calculation of the average age is still effectively iterative is obscured by the inherently recursive nature of the name collection.

As a final example, we consider a modification to the original `fat-cats` procedure. By a clever trick based on mutable pairs, it is possible to transform the linear recursive version into a linear iteration. The original version generates a recursive computation because no pair is allocated for `cons` until both of its subexpressions are fully evaluated. This means that the pairs composing the spine of the returned list are actually allocated in order from back to front, requiring the implementation to maintain a stack of pending `cons`es.

The trick is to allocate the pairs from front to back by using side effects. Here is a procedure that embodies this trick:[6]

```
(define (fat-cats-iter_mixed threshold database)
  (let ((ans-pair (cons 'ignore 'ignore)))
    (define (gather! record prev-pair)
      (if (end-of-database? record)
          (begin
            (set-cdr! prev-pair '())
            (cdr ans-pair))
          (if (> (record-get record 'salary)
                 threshold)
              (let ((next-pair (cons (record-get record 'name)
                                     'ignore)))
                (begin
                  (set-cdr! prev-pair next-pair)
                  (gather! (next-record record) next-pair)))
              (gather! (next-record record) prev-pair))))
    (gather! (first-record database) ans-pair)))
```

The iterative `gather!` procedure takes the previously allocated pair in addition to the current record. Every time a record satisfies the condition, it bashes the `cdr` of the previous pair to point to a newly allocated pair, and then passes the new pair to the recursive call. The initial pair passed to `gather!` is a dummy whose `cdr` will ultimately be the final answer list; `gather!` returns this answer list when the end of the database is reached.

This trick, which I will call *cdr-bashing list collection*, is clearly handy in many list processing programs. However, the monolithic approach does not allow us to package

[6]The resulting procedure is annotated with a $_{mixed}$ subscript because it combines aspects of both the functional and imperative styles.

this trick into a component that can be used elsewhere. In fact, the convoluted structure of `fat-cats-iter`$_{mixed}$ makes it difficult to even notice that it has been obtained from `fat-cats` by applying a trick!

2.2 Tree Example: Alpha Renaming

2.2.1 Overview

All the programs considered thus far generate linear computations. In this section, we will study a program that generates a tree-structured computation. Tree-shaped computations are important because, in the absence of control features like non-local exits and continuations, the procedure calls dynamically executed in any program can naturally be arranged in a tree. Linear computations are trivially a subset of tree computations. Moreover, we will see that tree-shaped computations support a richer set of evolution patterns than do linear computations.

The program we consider is *alpha renaming*, an operation on abstract syntax trees that is common in interpreters and compilers. Alpha renaming is one of the simplest practical programs that involves combining different shapes of tree walks in a nontrivial way.

Alpha renaming is a mechanism for consistently renaming the variables in a program so that each variable has a unique name.[7] This transformation is especially useful in lexically scoped languages, which permit the same identifier to name different logical variables within a single program. Many other expression transformations (e.g., substitution) are easier to perform on an expression that has been alpha renamed.

For simplicity, I will initially present alpha renaming in the context of an extremely simple language, the lambda calculus. The syntax of lambda calculus terms E is given by the following grammar:

[7] Technically, alpha renaming is any renaming transformation that maintains the same "connectivity" of variable declarations and references within a program. It does not necessarily imply making all variable names unique. However, in practice, the term is typically used to indicate that the resulting variable names are all distinct.

$$
\begin{array}{rcll}
E & ::= & I_{\text{use}} & \text{[Variable reference]} \\
 & | & \texttt{(lambda } I_{\text{def}} \ E_{\text{body}}\texttt{)} & \text{[Abstraction]} \\
 & | & \texttt{(call } E_{\text{rator}} \ E_{\text{rand}}\texttt{)} & \text{[Application]} \\
I & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{aa} \mid \texttt{ab} \mid \ldots &
\end{array}
$$

For our purposes, it is acceptable to view the lambda calculus as a restricted Lisp dialect where all procedures take exactly one argument and applications are tagged with an explicit `call` keyword. The only detail that matters for the present example is that the identifier I_{def} introduced by a `lambda` term declares a variable that can be referenced anywhere within E_{body} (as long as there is no intervening declaration of another variable with the same name).

The lambda calculus is about as spartan as a programming language can be. Later, we will consider extending it with extra features that make it more palatable to program in. We will see that a good test of the modularity of an alpha-renaming program is how little it needs to be changed in order to accomodate such features.

As an example of alpha renaming, consider the lambda calculus term:

```
(lambda x
  (lambda y
    (call (lambda x x)
          (call (lambda y x)
                (call (lambda x y)
                      z)))))
```

In this term, there are three logically distinct variables named `x`, two named `y`, and one named `z`. Variable references that occur within the scope of a declaration are said to be *bound*; those, like `z`, that are not in the scope of a declaration are said to be *free*.

The following is an alpha renamed instance of the above term:

```
(lambda x_1
  (lambda y_2
    (call (lambda x_3 x_3)
          (call (lambda y_4 x_1)
                (call (lambda x_5 y_2)
                      z)))))
```

Here, all declared variables and bound references have been consistently renamed by extending the original name with a unique number. (The free variable reference `z` cannot been renamed.) Any other method for consistent renaming that guarantees distinct names for distinct logical variables would also be acceptable.

The standard technique for alpha renaming can be visualized as the superposition of three tree-walking computations, as sketched in Figure 2.2:

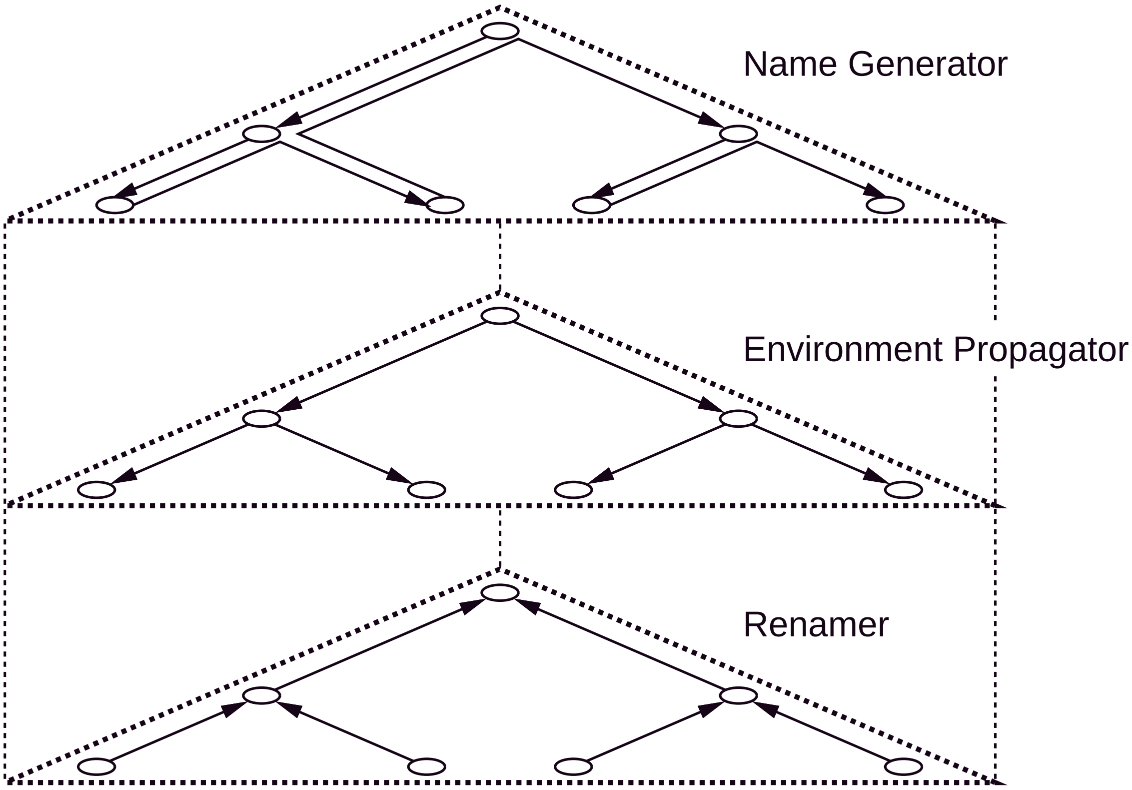


Figure 2.2: Sketch suggesting the different shapes the computations involved in alpha-renaming.

1. A *name generator* that creates a fresh name for every declared variable. To guarantee distinctness among the fresh names, the generator must conceptually carry with it some history of the names generated so far. Since the history at any point in time depends on the history at the previous point in time, the name generation computation effectively traces out a single-threaded dependence path through the syntax tree specified by the term. Whether the generator walks the term tree left-to-right, right-to-left, or by some other route is immaterial as long as the path is a single thread.

2. An *environment propagator* that transmits the fresh name for each declared variable down to all references of that variable. Since both subterms of an application see the same set of old-name/new-name bindings, the environment computation conceptually fans out into independent subcomputations at each `call` node. This downward fan shape contrasts with the single-threaded nature of the name generator.

3. A *renamer* that makes a copy of the term tree in which variable declarations and references have been replaced by the appropriate fresh name passed down by the environment propagator. The renaming computation conceptually starts at the leaves of the term tree and builds a new term on its way up the tree.

Although the above algorithm is standard, the form of the description — a decomposition into superposed computations — is not. One of the main themes of this dissertation is that conceptualizing programs in terms of interacting processes is essential to good modularity regardless of whether or not the programs are actually run on multi-processor machines. Each of the computations described above is a natural modular unit. Such computations are valuable building blocks for other programs; for example, environment propagating computations crop up all the time in interpreters and static analyzers. And such computations have the interchangeability usually associated with modules; for instance, in some contexts it might be worthwhile to replace a left-to-right name generator with a right-to-left one.

From this perspective, it seems reasonable to judge alpha-renaming programs by how well they reflect the modular structure described above. Unfortunately, the typical monolithic implementations exhibit little of this structure.

2.2.2 Monolithic Style: Functional Approach

Figure 2.3 presents an alpha renaming procedure written in the functional paradigm. The internal `walk` procedure takes three arguments: the term to be renamed; an environment that associates original variable names with their new names; and a positive integer representing a counter that is incremented every time a new variable declaration is encountered. The integer is used as an argument to `fresh-name`, which is responsible for generating unique names.[8] `Walk` returns two results: the alpha-renamed term, and the current value of the counter. Appropriate procedures for environment manipulation, fresh name generation, and syntactic abstraction (e.g., `lambda?`, `make-call`) are straightforward and have been omitted.

The `alpha-rename`$_{fun}$ procedure exhibits *none* of the modular structure suggested by Figure 2.2. Name generation, environment propagation, and construction of the renamed term are all inextricably intertwined in the single `walk` procedure. There is no handle for reusing one of the computations in a different program, or for easily modifying one of these computatations independently of the others. For example, the name generator hidden in the above program performs a left-to-right walk over the tree. What changes would be required to yield a right-to-left walk?[9] Answering this question requires consideration of the whole program rather than just a name generation piece. Furthermore, when making the changes, we must be careful to maintain the integrity of the other conceptual computations.

Finally, consider how `alpha-rename`$_{fun}$ would have to be modified in response to extending the base language with the conditional term `(if` E_{test} E_{then} E_{else}`)`. The dispatch within `walk` would need to be extended with a clause like the following:

```
((if? exp)
 (mlet (((new-test num1) (walk (test exp) env num)))
   (mlet (((new-then num2) (walk (then exp) env num1)))
     (mlet (((new-else num3) (walk (else exp) env num2)))
       (list (make-if new-test new-then new-else) num3)))))
```

This is an awfully complicated mess for something as simple as a conditional! The problem is that the bookkeeping details associated with returning both the renamed term *and* the updated counter obscure the following essential facts:

- The counter flows in a left-to-right fashion through the three subterms of the `if`.
- The environment flows unchanged into the three subterms of the `if`.

---

[8] For `alpha-rename`$_{fun}$ to be correct, it is necessary to assume that `fresh-name` returns a name disjoint from the set of free variables of the term.

[9] In the alpha renaming program, the order of traversal doesn't affect the correctness of the result. However, it's easy to imagine similarly structured programs in which some traversal orders are more desirable than others.

```
(define (alpha-rename_fun exp)

  (define (walk exp env num)
    (cond
     ((variable? exp)
      (list (env-lookup exp env) num))
     ((lambda? exp)
      (let ((old-formal (formal exp)))
        (let ((new-formal (fresh-name old-formal num)))
          (mlet (((new-body num1) (walk (body exp)
                                        (env-extend old-formal
                                                    new-formal
                                                    env)
                                        (+ num 1))))
            (list (make-lambda new-formal new-body) num1)))))
     ((call? exp)
      (mlet (((new-rator num1) (walk (rator exp) env num)))
        (mlet (((new-rand num2) (walk (rand exp) env num1)))
          (list (make-call new-rator new-rand) num2))))
     ))

  (mlet (((new-exp final-num) (walk exp env-standard 0)))
    new-exp))
```

Figure 2.3: Monolithic version of an alpha renamer written in the monolithic style.

Since these are the default flows associated with the name generation and environment computations, we would prefer a modularization in which we didn't have to specify them explicitly for the `if` case at all! After all, one of the measures of good modularity is the localizability of changes — internal modifications to one module shouldn't require changing connected modules.

2.2.3 Monolithic Style: Imperative Approach

Judicious use of side effects can improve the modularity of the alpha renaming program. Consider the `alpha-rename`$_{imp}$ procedure in Figure 2.4. The key difference between this

```
(define (alpha-rename_imp exp)

  (define gensym!
    (let ((num 0))
      (lambda (name)
        (let ((new-name (fresh-name name num)))
          (begin (set! num (+ num 1))
                 new-name)))))

  (define (walk exp env)
    (cond
     ((variable? exp) (env-lookup exp env))
     ((lambda? exp)
      (let ((old-formal (formal exp)))
        (let ((new-formal (gensym! old-formal)))
          (make-lambda new-formal
                       (walk (body exp)
                             (env-extend (formal exp)
                                         new-formal
                                         env))))))
     ((call? exp)
      (make-call (walk (rator exp) env)
                 (walk (rand exp) env)))
     ))

  (walk exp env-standard))
```

Figure 2.4: Monolithic version of an alpha-renamer written in the monolithic style.

procedure and the previous one is that most of the name generation computation has been captured in the `gensym!` procedure. The `gensym!` procedure owns a local state variable (`num`) that maintains the counter which was spread throughout the entire `walk` procedure within alpha-rename$_{fun}$. Because the single threading of the name generation computation is now being managed implicitly by side-effects rather than explicitly by data flow, the interface to the walk procedure is much simpler: it takes one less argument and returns one less result than in the functional version. This makes the resulting code easier to read and extend. For example, an if term can be handled by extending walk with the following clause, which is far simpler than the solution for the functional version:

```
((if? exp)
 (make-if (walk (test exp) env)
          (walk (then exp) env)
          (walk (else exp) env)))
```

This example underscores the importance of state and side effects as a technique for modularizing programs. This point is certainly not new and is elegantly argued elsewhere ([ASS85], [Bar92]). However, it is worth emphasizing that side effects are an important tool that cannot be neglected in our goal of modularizing programs.

In spite of the improvements, alpha-rename$_{imp}$ fails to achieve modularity in some important ways:

- The notion of a downward-flowing environment computation is still intertwined with the tree-construction computation in the single walk procedure. This means that the environment computation is not an entity that can be reused elsewhere. Furthermore, this organization forces the if handler to explicitly indicate that the environment is passed unchanged to each recursive call to walk on the subterms of the if. In an ideal scenario, this default flow shouldn't have to be explicit.

- More subtly, while *most* of the name generation computation has been localized to gensym, not *all* of it has. The order in which name generation visits the nodes of the term tree is inherited from the order in which the implementation language visits the arguments to a procedure call. In Scheme, where the order of argument evaluation is unspecified, no particular order can be relied upon. If for some reason we want to guarantee that the name generator visits subterms in left-to-right order, it is necessary to rewrite the call handler as:

```
((call? exp)
 (let ((new-rator (walk (rator exp) env)))
   (let ((new-rand (walk (rand exp) env)))
     (make-call new-rator new-rand))))
```

Here, Scheme's (strict) `let` construct forces the operator walk to be performed before the operand walk. A similar approach would have to be used for for cases like `if`.

Again, the order of the name generator doesn't matter in this particular problem, but traversal order does matter in other problems with a similar structure. In general, we would like a mechanism by which we can easily choose from options like left-to-right, right-to-left, and don't-care. A modularity technique providing finer control over the ordering of side effects (when it matters) is preferable to one that offers no control.

We have given numerous examples demonstrating that the monolithic approach fails to capture an important class of programming idioms in an effective way. It is obviously desirable to consider program organizations in which such idioms can be captured as modular units.

2.3 Slivers Capture Programming Idioms

2.3.1 Two Approaches to Decomposing Computations

We have seen that procedures like `mean-age`, `fat-cats`, and `alpha-rename` are conceptually composed out of many idiomatic units. But what is the nature of these units, and how are they combined to yield a program?

Viewing computations graphically gives some insight into these questions. Figure 2.5 shows a *computation diagram* for `fat-cats`. A computation diagram is a kind of circuit diagram for computations. In the diagram, every procedure application is represented as a labelled rectangular *device*, except for calls to `fat-cats` and `gather`, which have been expanded in terms of their definitions.[10] (Procedure calls represented by devices will be called *unexpanded*, while those replaced by the structure of their bodies will be called *expanded*.) Triangles pointing into a device are *input ports* that represent the procedure's arguments,

[10]In the diagram, `salary` and `name` are abbreviations for calls to `record-get`.

while triangles pointing out of a device are *output ports* for the procedure's results. Devices are connected by *wires* that specify the data dependences between procedures. Values can be thought of as flowing along the wires in the direction of the arrows.

This picture is similar to graphical depictions of dataflow programs [Den75, DK82] except that here the devices can compute only once, and the wires can transmit only a single value token before they are "used up". A computation diagram can be viewed as a dataflow diagram unwound so that every spatial entity is also associated with a unique point of time during the execution of the program. We will have much more to say about the static and dynamic properties of such computation diagrams in Chapter 8. For now, we will focus on ways in which the diagram can be decomposed into modular units.

The dashed boxes in Figure 2.6 indicate a *layer decomposition* of the computation diagram for `fat-cats`. Each box, called a *layer*, represents an expanded procedure call: the topmost box represents a call to `fat-cats`, while the other boxes represent calls to `gather`. Each layer contains devices for the unexpanded subcalls in the procedure's body, and sits on top of the layers for the expanded subcalls in the procedure's body. Layers are glued together by wires that pass arguments down to sublayers and receive results up from sublayers. Some of the arguments are explicit in the code (e.g., the results of the `next-record` devices correspond to the `record` parameter of `gather`) while others are implicit (the `threshold` "bus" down the middle of the diagram indicates that lexical scoping effectively makes `threshold` an implicit argument of `gather`).

Figure 2.7 shows a so-called *sliver decomposition* of the same diagram. Here the dashed boxes, called *slivers*, parse the diagram into vertical components rather than horizontal ones. The slivers are glued together by *cables*, collections of wires that pass information in and out of the sides of the slivers.

The rules for what comprises a legal sliver are very loose. The only real requirement is that if a sliver contains one instance of a device corresponding to a particular call in the procedure underlying the diagram, then it must contain all such instances. But beyond that, there are just some general guidelines for choosing slivers. The slivers are typically chosen to encapsulate repeated units of related functionality while minimizing the structure of the cables between them. All instances of the three devices for database manipulation

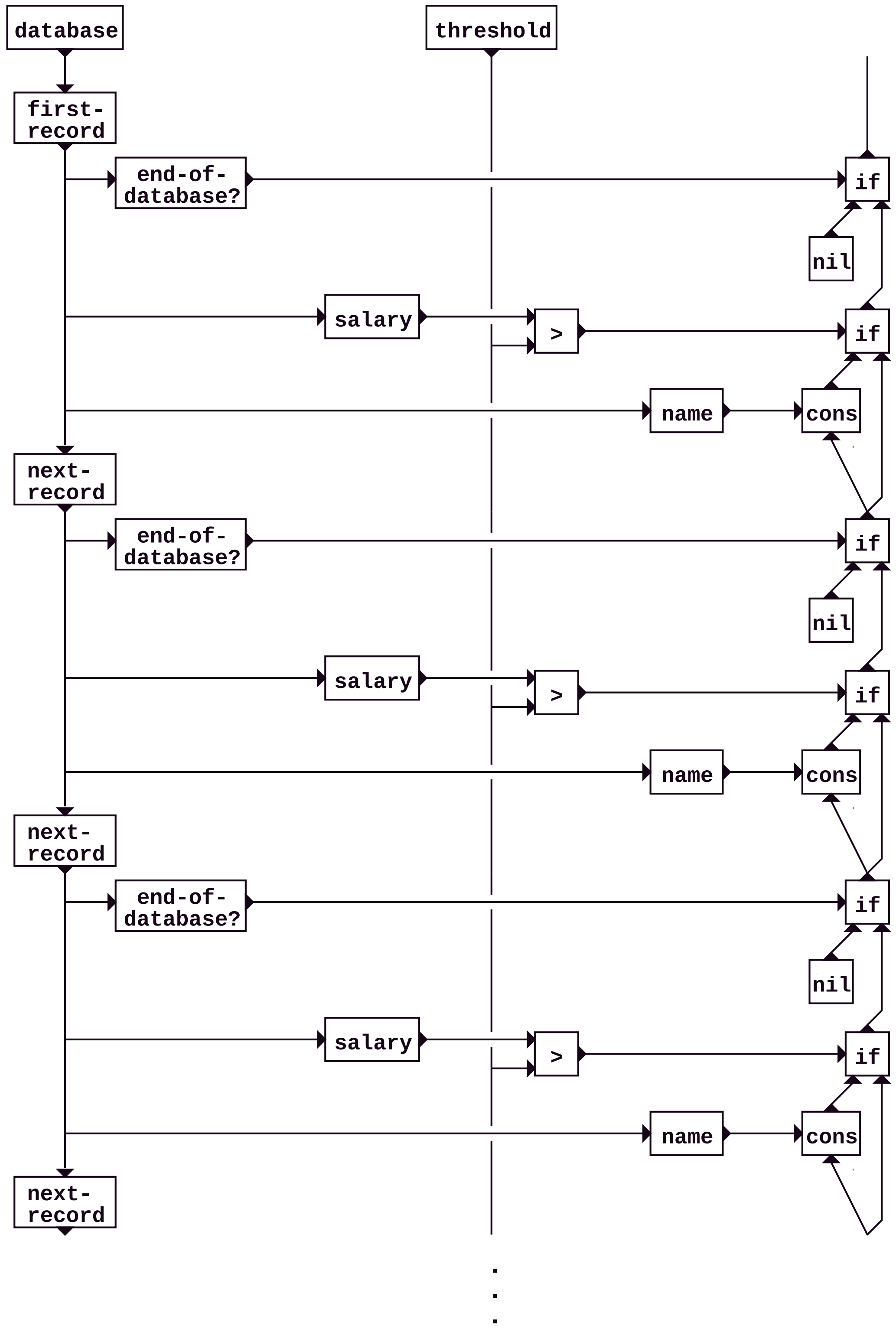


Figure 2.5: Computation diagram for `fat-cats`.

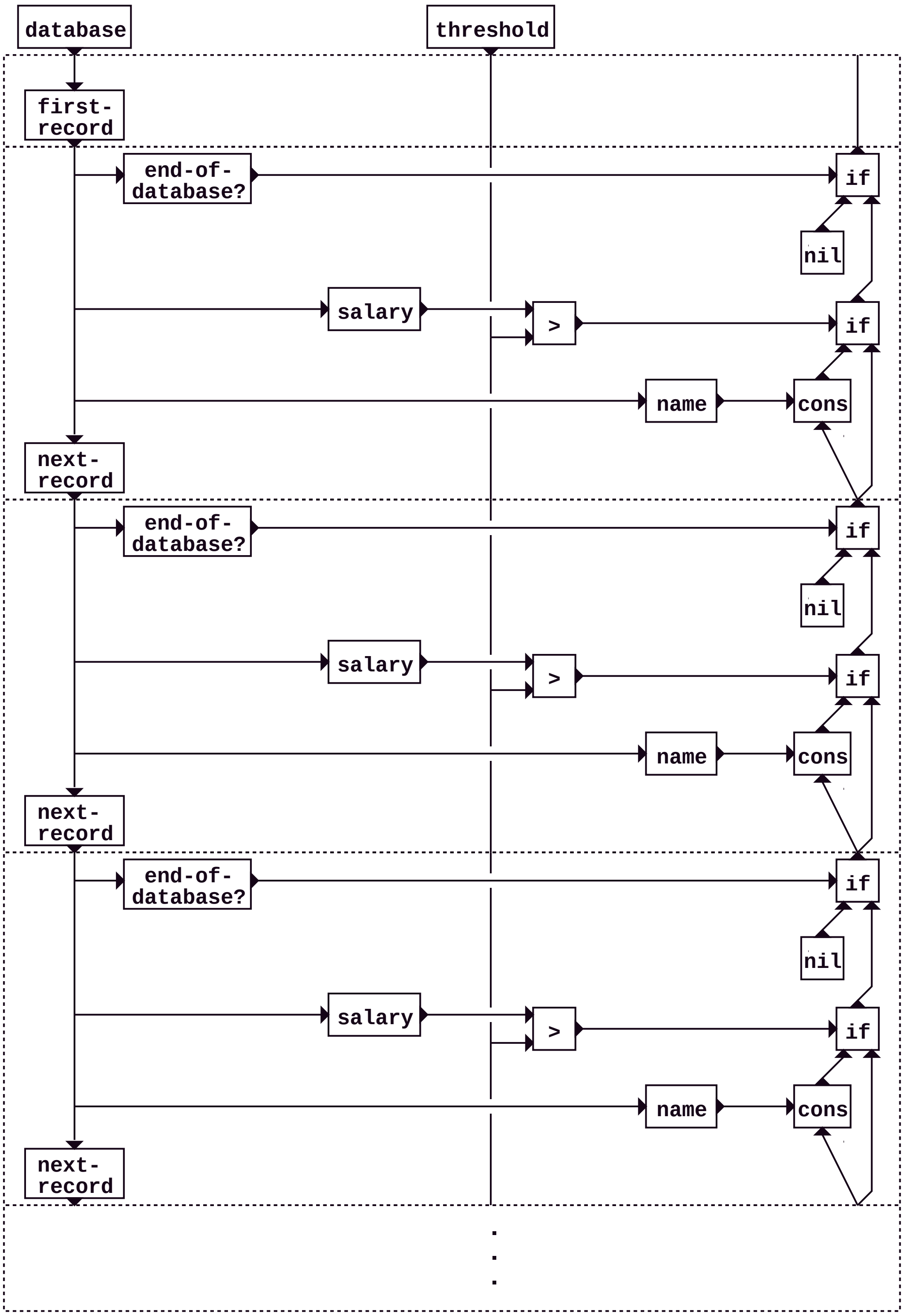


Figure 2.6: Layer decomposition of `fat-cats`.

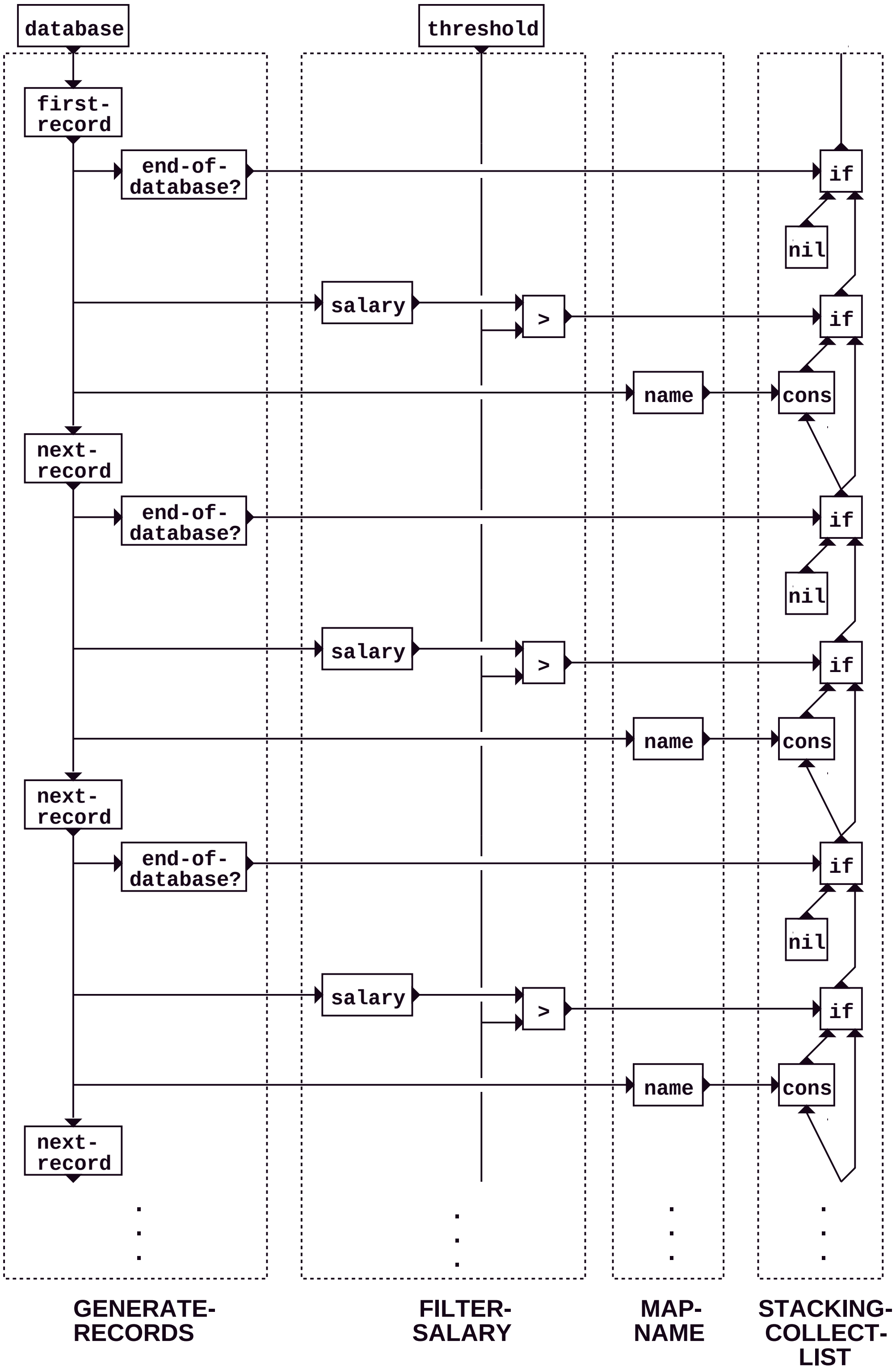


Figure 2.7: Sliver decomposition of `fat-cats`.

are bundled into the sliver labelled `GENERATE-RECORDS`, while the devices responsible for list collection are packaged up into `STACKING-COLLECT-LIST`. (The term "stacking" is intended to indicate that the sliver builds the resulting list from bottom up.) It would be possible to decompose these slivers up even further, say by factoring the `end-of-database?` devices or the `nil` devices into their own separate slivers, but this did not seem particularly desirable in this case. The devices within a sliver often form a single connected component of the computation diagram, but slivers like `FILTER-SALARY` and `MAP-NAME` consist of repeated connected components, all of which are mutually disjoint.

The most remarkable feature of the sliver decomposition in Figure 2.7 is that the individual slivers correspond closely to the kinds of idioms mentioned in the discussion of `mean-age` and `fat-cats`. The `GENERATE-RECORDS` sliver takes a database and spits out the individual records of the database in succession, along with a boolean *termination flag* indicating whether the database termination record has been found. Conceptually, this component occurs in both `mean-age` and `fat-cats`. The `STACKING-COLLECT-LIST` sliver accumulates elements into a list from the bottom up; elements are conditionally included in the resulting list depending on the value of an associated boolean *presence flag*. This can be made into a general linear recursive accumulator by abstracting the `cons` and `nil` to be any binary operator and base value. `FILTER-SALARY` effectively filters the records based on salary by providing the presence flag to the list collector, while `MAP-NAME` simply extracts the names from each record.

A host of other idioms can be depicted as slivers. For example, running sum and cdr-bashing list collection are two linear iterative accumulators that can be encapsulated into slivers. And the three tree-traversal idioms for `alpha-rename` sketched in Figure 2.2 can also be fleshed out into tree-shaped slivers.

What feature of sliver decompositions allows it to capture so many idioms? Whereas a layer decomposition focuses on the recursive pattern of a computation, a sliver decomposition focuses on recursionless operators. In effect, the recursion has been distributed over the slivers rather than being the main organizational principle for the program. The sliver-based organization directs attention away from the structural details of recursion and towards a more functional view of how program units fit together.

The notion of distributing loops over program components to enhance modularity is an old one. It has its origins in Lisp's higher-order list operators and APL's array operators, and is now used extensively as a technique in data parallel languages, functional and mostly functional languages, concurrent languages, and stream-based languages. We will study many of these techniques in detail in Chapter 3.

The new idea in slivers is to provide decompositions for a more general class of recursive computations than those handled by existing techniques. We shall see that most of the techniques alluded to above are limited to expressing linear iterative computations. Even the techniques that handle more general computations exhibit other limitations that constrain the class of decomposable computations. The goal of slivers is to express general tree-structured computations as compositions of mix-and-match parts. The strategy is to gain insight into the nature of these parts by studying sliver decompositions of monolithic computations that contain them.

Before we go on, it is worth pointing out that the sliver decomposition of Figure 2.7 is not an ideal modular decomposition. The problem is that the list collection sliver "knows" about the filtering sliver in the way that it handles the presence flag. In a sliver decomposition for a program that lists all employees, the list collection sliver would not handle any presence flags at all; for a similar program with two filtering predicates, the list collection sliver might need to handle two presence flags per record. It is clearly unreasonable to require different list collection slivers for each of these situations. In some sense, the instances of `if` that manipulate the presence flag really belong in the filtering sliver. But this change would greatly complicate the wiring between the filtering and accumulation slivers; it would add a cable loop between the slivers, and the clear upward flow of information in `STACKING-COLLECT-LIST` would disappear. In Chapter 5, we will investigate various methods for enhancing the modularity of computations that involve filtering.

2.3.2 Procedural Slivers

Just as it is important to distinguish procedures from the computations they specify, it is necessary to distinguish *procedural slivers* from *computational slivers*. A computational sliver is what we have simply been calling a sliver up to this point: a repeated pattern of devices in a computation diagram that embodies a programming idiom. A procedural sliver is a specification of this repeated pattern. Intuitively, it is that fragment of a procedure that encapsulates the idiom.

How can such a fragment be specified? Investigating answers to this question is one of the main goals of this report. For the moment, the following approach may be helpful for envisioning procedural slivers. Imagine starting with the text of a procedure and erasing all the program structure that is not relevant for generating a given computational sliver. Then what's left over is a crude kind of procedural sliver.

Figure 2.8 shows the descriptions that result from using this approach for three of the computational slivers of `fat-cats`. The resulting procedural slivers don't make much sense individually, and certainly aren't executable. However, a sensible program like `fat-cats` can be constructed by textually overlaying its component procedural slivers. In computation space, the textual overlaying of procedural slivers corresponds to combining computational slivers in a side-by-side manner to yield the computation diagram for the entire procedure. The challenge is to develop a programming model in which the specification and combination of procedural slivers is no harder than procedure specification and combination.

For the remainder of this report, we will loosely use the term *sliver* for both computational slivers and procedural slivers when the meaning is clear from context or the distinction doesn't matter. The modifiers "computational" and "procedural" will only be sprinkled in when we wish to emphasize the distinction.

2.3.3 Sliver Diagrams

Sliver decompositions can be abstracted into *sliver diagrams* that summarize the sliver and cable interconnections while suppressing many details. Figure 2.9 shows one possible diagram for the `fat-cats` sliver decomposition. Each sliver is represented as a box, while cables are represented as thick, directed conduits between slivers. Individual devices and wires are represented as before.

The sliver diagram in Figure 2.9 is a very literal interpretation the sliver decomposition in Figure 2.7 in the sense that the slivers and cables are assumed to have exactly the structure shown in the decomposition. For example, in Figure 2.9, the input cable to the

```
;;; Fragment responsible for GENERATE-RECORDS
;;;
(define (fat-cats           database)
  (define (gather record)
        (end-of-database? record)






            (gather (next-record record)))
  (gather (first-record database)))

;;; Fragment responsible for FILTER-SALARY.
;;; (The fragment for MAP-NAME is similar)
;;;
(define (fat-cats threshold         )
  (define (gather       )


            (> (record-get record 'salary)
               threshold)



            (gather                     ))
  (gather                           ))

;;; Fragment responsible for STACKING-COLLECT-LIST
;;;
(define (fat-cats                  )
  (define (gather       )
    (if
        '()
        (if

            (cons
                  (gather                     )
            (gather                     )))))
  (gather                           )
```

Figure 2.8: Procedural fragments approximating the procedural slivers that generate the computational slivers of `fat-cats`. These are obtained by "whiting out" structure that does not appear in a given computation sliver. The original program can be obtained by textually overlaying all of its component fragments.

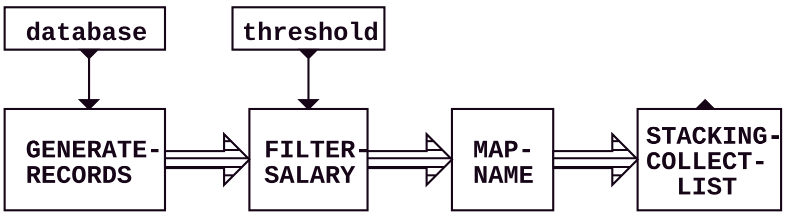


Figure 2.9: A sliver diagram summarizing the sliver decomposition for `fat-cats`.

`MAP-NAME` sliver is assumed to carry three types of wires:

1. *Termination wires* that transmit a boolean indicating whether the end of the database has been reached.

2. *Presence wires* that transmit a boolean indicating whether the filter has passed the associated record.

3. *Element wires* that transmit the current record when the corresponding terminations and presence wires both transmit a true value.

The output cable of the `MAP-NAME` sliver also carries the same three types of wires. `MAP-NAME` passes the termination and presence wires unchanged, but transmits a string rather than a record on the element wire.

However, not every wire that happens to pass across a vertical dotted box in a sliver decomposition need be considered part of that sliver. Figure 2.10 depicts a sliver diagram for a different perspective on the sliver decomposition of `fat-cats`. Here, the cable produced by the record generator fans out and feeds both `FILTER-SALARY` and `MAP-NAME`, while the list collector now takes two input cables:

1. A cable of presence wires from the salary comparison.

2. A cable of element wires from the name mapper.

(Presumably one or both of these cables also carries the termination wires.) It is easy to imagine yet other diagrams in which each type of wire is carried by a distinct cable. Of course, the interfaces to the abstracted slivers depends on the chosen interpretation. The challenge is to design the diagrams to maximize the reusability of the abstracted slivers. We will discuss this in detail later; for now, we will choose Figure 2.9 as the "standard" sliver diagram for `fat-cats`.

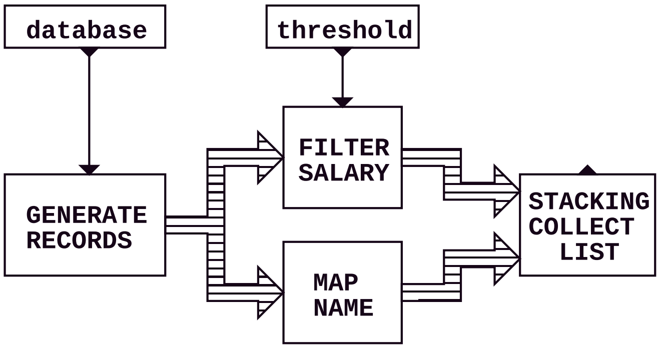


Figure 2.10: An alternate sliver diagram for `fat-cats`.

Sliver diagrams underscore the advantages of capturing idioms in modular units. They provide a convenient framework in which to understand and compare programs. For example, the block diagram in Figure 2.11 elucidates the structure of `mean-age`. The fact that there are two running sums is obvious from the structure of the diagram. The `MAP-ONE` sliver maps every input record into the constant `1`; summing these `1`s up gives the employee count. From the sliver diagrams, it is easy to see that `mean-age` uses the same database enumeration strategy as `fat-cats`.

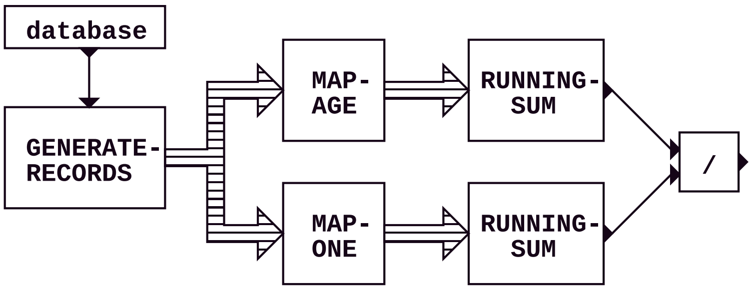


Figure 2.11: A sliver diagram for `mean-age`.

Perhaps the biggest advantage of sliver diagrams is that they promote the notion that programs should be organized out of mix-and-match parts. For example, Figures 2.12

and 2.13 support the intuition that the `fat-cat-age` and `fat-cats&mean-age` procedures are essentially series and parallel combinations of `fat-cats` and `mean-age`.[11] Replacing the recursive `STACKING-COLLECT-LIST` sliver in Figure 2.9 with the iterative `CDR-BASHING-COLLECT-LIST` sliver (Figure 2.14) represents the constant-space computation generated by the `fat-cats-iter` procedure. It is apparent from the diagrams that the iterative version is obtained from the recursive one by a simple modification; moreover, that modification is encapsulated in a way such that it can easily be plugged in elsewhere. Components like `RUNNING-SUM`, `STACKING-COLLECT-LIST`, and `CDR-BASHING-COLLECT-LIST` are especially reusable because they do not depend on particular details of the database example.[12]

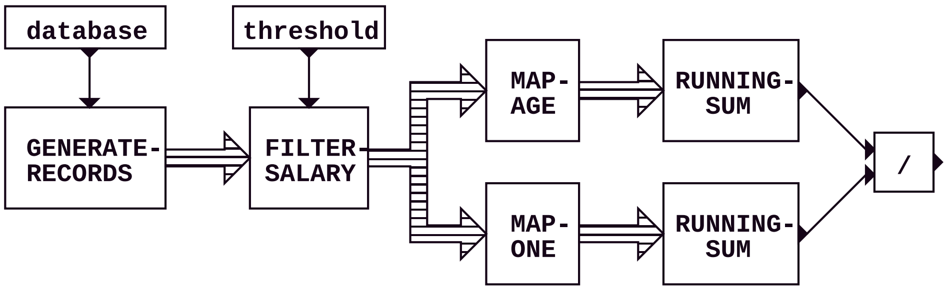


Figure 2.12: A sliver diagram for `fat-cat-age`.

As a final example, Figure 2.15 shows a sliver diagram for the tree-based `alpha-rename` computation. Here, each cable transmits tree-structured information between the slivers. The name generation, environment propagation, and renaming computations have each been encapsulated in their own sliver. There are some additional slivers, though. The `FILTER-FORMALS` sliver finds the names of the declared variables; this has been factored out of the name generation computation so that the names can be supplied to the environment computation as well. And the `TERM->TREE` and `TREE->TREE` slivers provide conversions between concrete terms and the "exploded" versions of their abstract syntax trees that are

[11] Figure 2.12 is somewhat misleading, since it uses versions of `MAP-AGE`, `MAP-ONE`, and `RUNNING-SUM` that must handle the presence flag produced by `FILTER-SALARY`. Yet, the corresponding slivers in Figure 2.11 do not manipulate a presence flag. This is another instance of the problems inherent in filtering that we will need to deal with later.

[12] Modulo the above-mentioned issues of filtering.

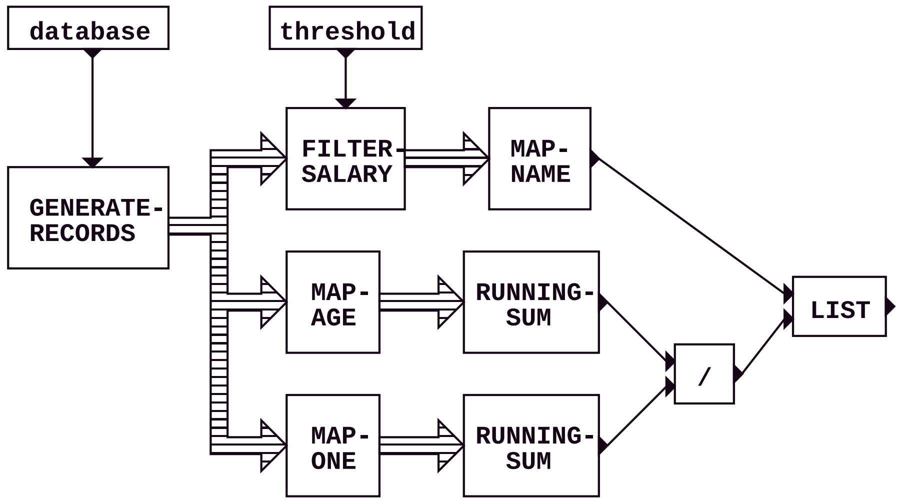


Figure 2.13: A sliver diagram for `fat-cats&mean-age`.

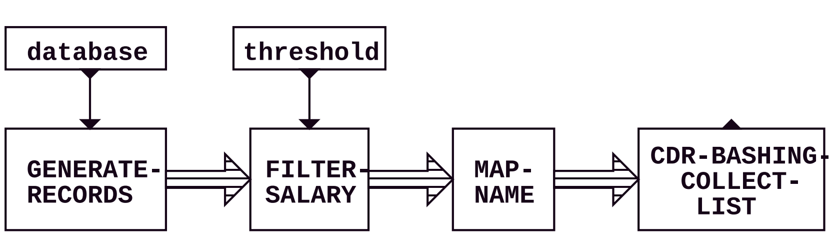


Figure 2.14: A sliver diagram for `fat-cats-iter`.

transmitted via the cables.[13]

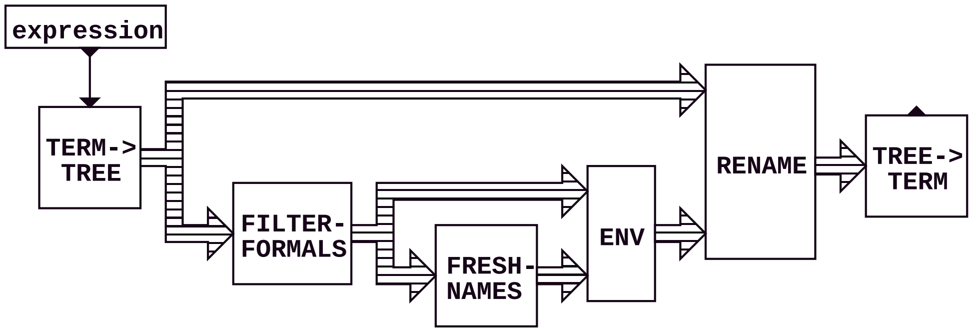


Figure 2.15: A sliver diagram for `alpha-rename`.

A key property of sliver diagrams is that they explicitly show only the flow of data, not of control. As noted above, an upshot of modularizing the kinds of idioms discussed above is that loops and recursions are distributed over individual devices rather than being the main organizing principle for programs. This approach directs attention away from the often complex details of control flow and towards a more functional view of how program units fit together. Reasoning about control is localized to the implementations of individual devices, where it is much more tractable.

2.3.4 Operational Interpretation of Sliver Diagrams

The previous examples demonstrate that it's easy to combine slivers in mix-and-match ways. But the operational behavior of the programs specified by the resulting sliver diagrams is rather ambiguous. Suppose we presented the sliver diagrams in Figures 2.9 – 2.15 to other programmers and described the high-level purpose of each box but did not explain how the diagrams were derived from monolithic programs. Then the programmers could easily construct many distinct but consistent stories about the meaning of these diagrams that were different from our intended meaning. Here are some alternate interpretations:

1. The boxes are procedures that take and return aggregate data structures.

[13]This is only true as long as there are no circularities in the data dependencies implied by the cables. While such "data loops" can be be useful programming techniques (e.g. [Bir84]), they can also complicate reasoning about programs by requiring programmers to think in terms of fixed points.

2. The boxes are demand-driven agents that request individual values from and return individual values to their neighbors.

3. The boxes are data-driven processes that concurrently consume and produce values sent over communicating channels.

These interpretations correspond to some of the classical techniques for achieving modularity by distributing loops across programming idioms.

But the existence of so many interpretations means that sliver diagrams fail to nail down important operational characteristics of computations. In the case where sliver diagrams are derived from monolithic procedures, we'd like the resulting diagrams to specify operational behavior similar to the original procedures. And when the diagrams aren't derived from a single procedure, but are pieced together from existing slivers, we want a theory that defines how the operational behavior of the composite structure is determined from the operational behavior of the parts.

Exactly what is meant by "similar operational behavior" will necessarily remain vague until we present a formal model of computation (Chapter 8). But here we can at least give a few examples of what we have in mind:

- The monolithic `mean-age` and `fat-cat` procedures perform a single traversal over the database, while the monolithic `alpha-rename` procedure performs a single traversal over its argument expresssion. We would like the computation described by the corresponding sliver diagrams to maintain this single-traversal property.

- The monolithic `mean-age` procedure generates a computation that requires constant data and control space. The corresponding sliver diagram should also specify a constant-space computation.

- The monolithic `fat-cats` procedure does not compute the name for a record that does not pass the salary filter. In this case, the name computation is trivial, but it's easy to imagine cases where it is important to avoid computations on items that do not pass a filter. It is desirable for the computations specified by the sliver diagrams in both Figures 2.9 and 2.10 to avoid unnecessary operations. (The fan-out in Figure 2.10 makes it tricky to implement this behavior.)

In a truly modular approach to these operational concerns, the operational behavior of a computation should be derivable from operational behavior of its component parts. In order to capture operational nuances, it will be necessary to include operational details in the interfaces to slivers. We have done this to some extent in terms of the names of the list collection slivers; `STACKING-LIST-COLLECT` and `CDR-BASHING-LIST-COLLECT` have the same input/output behavior, but the first requires a control stack while the second does not. We'd like to have a method of reasoning about these kinds of space issues based on the structure of the slivers. Chapters 4 and 5 develop such a method. But first, in Chapter 3, we will explore what's wrong with existing methods for expressing sliver diagrams as programs.

Chapter 3

The Signal Processing Style of Programming

The *signal processing style (SPS)* of programming is a label for the class of techniques that organize programs like the sliver diagrams introduced in the previous chapter. In this style, computations are are expressed as networks of computational devices that generate, map, filter, and accumulate data transmitted over directed cables. This style encompasses a wide range of programming techniques used in functional, imperative, object oriented, and concurrent languages. The name "signal processing style" is suggested by the resemblance between sliver diagrams and signal processing block diagrams.

This chapter explores the tension between modularity and control in the signal processing style of programming. It elucidates two key points about this style:

- The signal processing style is a powerful means of decomposing programs into modular units that encapsulate important programming idioms.
- Classical techniques for programming in the signal processing style often preempt the programmer from controlling important operational aspects of programs expressed in this style (e.g., execution time, space complexity, operation scheduling).

The remainder of this dissertation explores ways of reducing the tension between modularity and control. In particular, the lock step processing model developed later is able to achieve modularity while preserving space complexity and operation scheduling (it does not address the control of execution time).

I will classify SPS techniques into two categories according to how they represent the slivers and cables in a sliver diagram:

1. The *aggregate data approach* represent slivers as operators that manipulate aggregate data (e.g., lists, streams, arrays, trees) and cables as the aggregate data structures themselves.

2. The *channel approach* represent slivers as communicating processes and cables as the communication channels between processes.

The lines between these approaches are not rigid; we will encounter techiques that exhibit characteristics of both.

In the following sections, we study these two approaches in the context of the database and alpha renaming examples. We will also consider several other techniques that resist classification into the above two categories. We will see that each of the standard SPS techniques is a two-edged sword: it helps the programmer subdue complexity, but it also either (1) prevents the programmer from controlling important operational behavior or (2) unduly limits the class of programs that can be expressed.

3.1 The Aggregate Data Approach

A common technique for encoding sliver diagrams is to represent slivers as procedures that manipulate aggregate data structures. Then the slivers can be wired together simply by the usual methods for procedural composition. For example, the following procedures are the textual encodings of the sliver diagrams for `fat-cats` (Figure 2.9) and `mean-age` (Figure 2.11):

```
(define (fat-cats database)
  (stacking-collect-list
   (map-name
    (filter-salary threshold
                   (generate-records database)))))

(define (mean-age database)
  (let ((records (generate-records database)))
    (/ (running-sum (map-age records))
       (running-sum (map-one records)))))
```

Here we assume that each procedure called in the body corresponds to the similarly-named sliver. Note how the wire connections are represented by the natural data dependences of nested subexpressions; in some sense, the aggregate values *are* the cables. Cable fan-out is handled by `let`, whose bindings allow the same aggregate value to be used by more than one procedure.

The aggregate data style has its roots in Lisp's list manipulation routines (as epitomized by `mapcar`) and APL's array operators. Today, the aggregate data approach is the main organizing principle for data parallel languages (e.g., Fortran 90 [Ame89], C* [RS87], NESL [Ble92], paralations [Sab88]). It is also a commonly used technique in many other languages, especially functional and mostly functional ones (e.g., Haskell [HJW$^+$92], Id [AN89], Common Lisp [Ste90], Scheme [CR$^+$91], ML [MTH90]).

3.1.1 Database Example: A List Implementation

For the linear database example, lists are an obvious choice for the type of aggregate data structure. Figure 3.1 shows how each of the slivers can be represented as a list-manipulation procedure.

Expressing `mean-age` and `fat-cats` as combinations of parts that share a standard interface (lists) is a powerful strategy because the parts are reusable in a mix and match way. Even though the code size for the aggregate data versions of these two programs (including the definitions of their subroutines) is much larger than the size of the monolithic versions, this increased size is offset by the modularity of the structure and the fact that the size of the subroutines should be be amortized over all the places where those subroutines will be used. Parts like those in Figure 3.1 are likely to be used pervasively in database applications. And parts like `map-one`$_{list}$ and `running-sum`$_{list}$ have even broader applicability, since they

```
;;; Generators
(define (generate-records_list database)
  (define (gen record)
    (if (end-of-database? record)
        '()
        (cons record (gen (next-record record)))))
  (gen (first-record database)))

;;; Mappers
(define (map-age_list records)
  (if (null? records)
      '()
      (cons (record-get (car records) 'age)
            (map-age_list (cdr records)))))

(define (map-one_list list)
  (if (null? list)
      '()
      (cons 1 (map-one_list (cdr list)))))

(define (map-name_list records)
  (if (null? records)
      '()
      (cons (record-get (car records) 'name)
            (map-age_list (cdr records)))))

;;; Filters
(define (filter-salary_database threshold records)
  (cond ((null? records) '())
        ((> (record-get (car records) 'salary) threshold)
         (cons (car records)
               (filter-salary_database threshold (cdr records))))
        (else (filter-salary_database threshold (cdr records)))))

;;; Accumulators
(define (running-sum_list list)
  (define (accum lst sum)
    (if (null? lst)
        sum
        (accum (cdr lst) (+ (car lst) sum))))
  (accum list 0))

(define (stacking-collect-list_list lst) lst) ; Already a list!
```

Figure 3.1: List-based implementation of the slivers used by `mean-age` and `fat-cats`.

will work on any linear structures, not only databases. This underscores the key advantage of the aggregate data approach: aggregate data operators constitute a *language* for working in a domain.

Using higher-order procedures, it is possible to obtain modular pieces boasting even greater reusability. Figure 3.2 introduces higher-order procedures that capture the essence of generation, mapping, and iterative accumulation. The `mean-age` subroutines are just instances of these more general abstractions (Figure 3.3).

```
(define (generate_list done? next current)
  (if (done? current)
      '()
      (cons current (generate_list done? next (next current)))))

(define (map_list function lst)
  (if (null? lst)
      '()
      (cons (function (car lst))
            (map_list function (cdr lst)))))

(define (filter_list predicate lst)
  (cond ((null? lst) '())
        ((predicate (car lst))
         (cons (car lst) (filter_list predicate (cdr lst))))
        (else (filter_list predicate (cdr lst)))))

(define (iter-accumulate_list operator identity list)
  (define (acuum lst ans)
    (if (null? lst)
        ans
        (accum (cdr lst) (operator (car lst) ans))))
  (accum list identity))
```

Figure 3.2: Higher-order list-manipulation procedures.

It is worth noting how the list-based implementation finesses the modularity problems with filtering for the sliver decomposition of Figure 2.7. A list representing a cable does not explicitly represent any presence flags. Rather, any element whose associated presence flag is false is simply not included in the list. This *compression* technique enhances modularity because there is no need to have different procedures for handling filtered vs. unfiltered data. Unfortunately, the compression trick does not extend to the tree-structured case; removing an element from a list leaves a list, but removing an element from a tree does not

```
(define (generate-records_list database)
  (generate/list end-of-database? next-record (first-record database)))

(define (map-age_list records)
  (map/list (lambda (rec) (record-get rec 'age)) records))

(define (map-one_list list)
  (map/list (lambda (rec) 1) list))

(define (running-sum_list list)
  (iter-accumulate/list + 0 list))
```

Figure 3.3: Implementations of the `mean-age` subroutines in terms of the higher-order list-manipulation procedures.

leave a tree! Later, we will see how to deal with this problem.

As an aside, it should be mentioned that the list-based programs are really only simulating the sliver diagrams for `mean-age` and `fat-cats`. Indeed, each of the procedures in Figure 3.1 can be described by its own sliver diagram! For example, the structure of `fat-cats`$_{list}$ is really as shown in Figure 3.4. Here, each of the four component procedures is associated with a dotted box containing the sliver diagram for that procedure. The core slivers (`GENERATE-RECORDS`, `FILTER-SALARY`, `MAP-NAME`) are wrapped in instances of `SPLAY-LIST` and `GLOM-LIST` that convert between the standard external list representation and the internal cable representation: `SPLAY-LIST` is a generator that, given a list, spits out its elements in order to a cable; `GLOM-LIST` is just a synonynm for `STACKING-LIST-COLLECT`. (`SPLAY-LIST` and `GLOM-LIST` are the analogues of buffers in electrical systems that present standard impedances to the rest of a circuit.) Each dotted box simulates the input/output behavior of the corresponding sliver in Figure 2.7, while each wire between dotted boxes carries a list that encodes an entire cable structure between slivers.

It is possible to imagine doing algebra on slivers. For example, `SPLAY-LIST` and `GLOM-LIST` act as inverses, so it should be possible to treat any juxtaposition of these two slivers as an identity. Using this fact in conjunction with the equivalence of `GLOM-LIST` and `STACKING-LIST-COLLECT`, it is easy to show that the diagram in Figure 3.4 is "equivalent" to the diagram in Figure 2.9. But this notion of equivalence only captures the input/output behavior of `fat-cats`. It does not capture how `fat-cats`$_{list}$ builds and takes

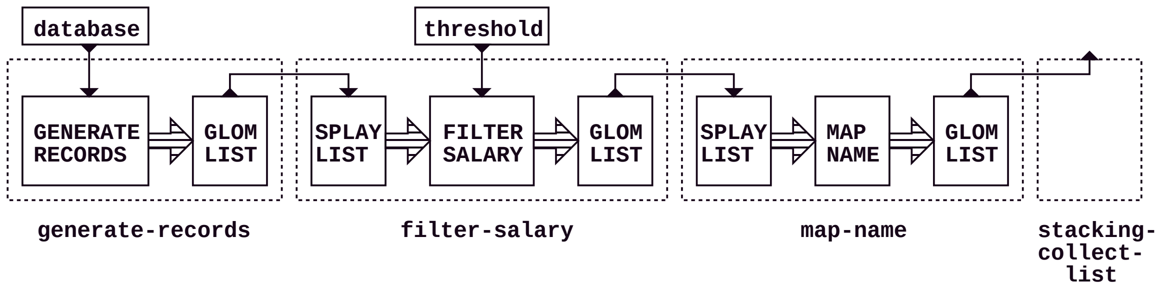


Figure 3.4: Sliver diagram for the list-based implementation of `fat-cats`. Each of the dotted boxes corresponds to the procedure whose name appears below it.

aparts intermediate lists that do not exist in $\texttt{fat-cats}_{fun}$. Nor does it capture how (in Scheme) $\texttt{fat-cats}_{list}$ performs all generation steps before any filtering steps (in contrast, $\texttt{fat-cats}_{fun}$ interleaves the steps from these stages).

3.1.2 Database Example: An Array Implementation

Lists are not the only data structure that can be used to implement the database examples in the aggregate data style. In a data-parallel language, the natural representation for cables would be some sort of array. In such a language, the basic strategy for the database examples would be to fill an array with all the database records and then use appropriate data-parallel array operators to derive a result.

A notable feature of data-parallel operators is that they tend to operate in an elementwise fashion on the input array. In this paradigm, a natural way to do filtering is to first compute a boolean array containing presence flags for the corresponding elements of an input array, and later use the presence flags to select the array indices at which an operation will be performed. We will call this approach the *gap* technique of filtering, because it models the absence of an element with an explicit marker indicating that the element is not there. Via the selection mechanism, all array operators are equipped to handled filtered data at all times; unfiltered data is just the special case where all indices are selected. Thus, modularity is enhanced in the gap technique by having every sliver handle filtered data.

3.1.3 Alpha Renaming Example: A Tree Implementation

Significant gains in modularity can be achieved by expressing alpha renaming as an SPS program. The sliver diagram in Figure 2.15 is easy to express in the aggregate data approach:

```
(define (alpha-rename_tree exp)
  (let ((exp-tree (splay-tree exp)))
    (let ((def-tree (filter-formals exp-tree)))
      (glom-tree (rename exp-tree
                         (environment def-tree
                                      (fresh-names def-tree)))))))
```

Since each of the conceptual processes involved in alpha renaming is some sort of tree walk, it is natural to use trees as the common currency through which the slivers communicate. To simplify the implementation, all of the intermediate trees have a common form. They are instances of a tree data abstraction in which every tree node is an immutable structure maintaining a single value (the *datum*) and a list of subtrees. The `tree` constructor takes a datum and list of subtrees and returns a newly constructed tree node with this information; the `tree-datum` and `tree-subtrees` selectors extract information from a tree. Here is a straightforward implementation of this abstraction that we will assume in our examples.

```
(define (tree value subtrees) (cons value subtrees))

(define (tree-datum tr) (car tr))

(define (tree-subtrees tr) (cdr tr))
```

The datum stored at a tree node can be any value. Multiple values can be stored at a tree node by packaging them up into a single compound datum. A tree whose subtree list is empty is called a *leaf*.

Figures 3.5 – 3.7 present the tree-based implementations of the subroutines used by `alpha-rename`$_{tree}$. Although lambda calculus terms have an inherent tree structure, it is still necessary to convert them into the common tree format used by the other operations. We will refer to the original syntactic form as an *term* and the converted one as a *node tree*. The conversion between terms and node trees is handled by `splay-tree` and `glom-tree`, as shown in Figure 3.5. The datum of an node tree, called a *node*, is a list of a term type and

```
(define (splay-tree exp)
  (cond
   ((variable? exp)
    (tree (list 'variable exp) '()))
   ((lambda? exp)
    (tree (list 'lambda (formal exp))
          (list (splay->tree (body exp)))))
   ((call? exp)
    (tree (list 'call)
          (list (splay->tree (rator exp))
                (splay->tree (rand exp)))))
   ))

(define (glom-tree tr)
  (let ((exp (tree-datum tr))
        (subexps (tree-subtrees tr)))
    (case (first exp)
     ((variable) (second exp))
     ((lambda)
      (make-lambda (second exp)
                   (glom-tree (first subexps))))

     ((call)
      (make-call (glom-tree (first subexps))
                 (glom-tree (second subexps))))
     )))

(define (filter-formals exp-tree)
  (let ((exp (tree-datum exp-tree))
        (subexps (tree-subtrees exp-tree)))
    (if (eq? 'lambda (first exp))
        (tree (second exp)
              (list (filter-formals (first subexps))))
        (tree '() ; No names declared at this node.
              (map filter-formals subexps)))))
```

Figure 3.5: Subroutines for the alpha-renamer, part I.

extra information relevant to that type. The subtrees of an node tree are just the converted subterms.

For instance, consider the following sample term:

```
(define a-term
  '(lambda x
     (lambda y
       (call (lambda x
               x)
             (lambda y
               x)))))
```

The node tree corresponding to `a-term` is given below:

```
(splay-tree a-term)
```
$\stackrel{\text{eval}}{\Longrightarrow}$
```
((lambda x)
 ((lambda y)
  ((call)
   ((lambda x)
    ((variable x)))
   ((lambda y)
    ((variable x))))))
```

The `filter-formals` procedure (see Figure 3.5) highlights the locations of the declared names in an expression tree. It returns a *name tree* with the same shape as its input in which every `lambda` node has been mapped to its formal parameter and every non-lambda node has been mapped to nil. This is an example of the gap technique of filtering applied to trees; here, a gap is explicitly represented by a nil. For example,

```
(filter-formals (splay-tree a-term))
```
$\stackrel{\text{eval}}{\Longrightarrow}$
```
(x
 (y
  (()
   (x
    (()))
   (y
    (()))))))
```

The name generator, `fresh-names` (see Figure 3.6), is the most complicated component in this decomposition. It transforms a given name tree into a new name tree in which all the names are distinct. It does this in two stages. First, it performs a left-to-right preorder walk over the given name tree, in which it increments a counter every time it encounters a name. This process returns a *number tree* with the same shape as the name tree in which every node is decorated with the value of the counter from the walk. Second, it maps a new-symbol generator over the name tree and the number tree to get a tree of fresh names. For example:

```
(lr-pre-number (filter-formals (splay-tree a-term) 0))
```
$\overset{\text{eval}}{\Longrightarrow}$
```
(0
 (1
  (2
   (2
    (3))
   (3
    (4)))))
```

```
(fresh-names (filter-formals (splay-tree test)))
```
$\overset{\text{eval}}{\Longrightarrow}$
```
(x_0
 (y_1
  (()
   (x_2
    (()))
   (y_3
    (())))))
```

It is worth noting that `lr-pre-number` works on any tree, not only name trees.

The `environment` procedure (Figure 3.7) takes a name tree and a value tree and creates a tree of environments that has the same shape as the two input trees. The environment datum at every node in the resulting tree is the environment of the parent node extended by a binding between the corresponding name and value. When there is no corresponding name (i.e., the name tree has a nil at a node), the environment is passed down unchanged from above. If we assume that environments are implemented as association lists (where earlier bindings take precedence over later ones), then `environment` has the following behavior on our test expression:

```
(let ((defs (filter-formals (splay-tree a-term))))
  (environment defs
               (fresh-names defs)))
```
$\overset{\text{eval}}{\Longrightarrow}$
```
(((x . x_0))
 (((y . y_1) (x . x_0))
  (((y . y_1) (x . x_0))
   (((x . x_2) (y . y_1) (x . x_0))
    (((x . x_2) (y . y_1) (x . x_0))))
   (((y . y_3) (y . y_1) (x . x_0))
    (((y . y_3) (y . y_1) (x . x_0)))))))
```

```
(define (fresh-names name-tree)
  (map-fresh name-tree
             (lr-pre-number name-tree 0)))

(define (lr-pre-number tr num)
  ;; Create a number tree for the given tree
  ;; by a left-to-right preorder walk.
  (define (walk tr num)
    ;; Returns a list of a new tree and a new num
    (mlet (((num-subtrees num1)
            (walk-trees (tree-subtrees tr)
                        (if (null? (tree-datum tr))
                            num
                            (+ num 1)))))
      (list (tree num num-subtrees)
            num1)))
  (define (walk-trees trs num)
    ;; Returns a list of a new tree list and a new num
    (if (null? trs)
        (list '() num)
        (mlet (((first-tree num1) (walk (car trs) num)))
          (mlet (((rest-trees num2) (walk-trees (cdr trs) num1)))
            (list (cons first-tree rest-trees)
                  num2)))))
  (car (walk tr 0)))

(define (map-fresh name-tree num-tree)
  (let ((name (tree-datum name-tree))
        (num  (tree-datum num-tree)))
    (tree (if (null? name)
              '()
              (fresh-name name num))
          (map map-fresh
               (tree-subtrees name-tree)
               (tree-subtrees num-tree)))))
```

Figure 3.6: Subroutines for the alpha-renamer, part II.

```
(define (environment name-tree val-tree)
  ;; Assume ENV-IDENTITY binds each name to itself
  ((env-down env-identity) name-tree val-tree))

(define (env-down env)
  (lambda (ntree vtree)
    (let ((name (tree-datum ntree))
          (val  (tree-datum vtree)))
      (let ((new-env (if (null? name)
                         env
                         (env-extend name val env))))
        (tree new-env
              (map (down new-env)
                   (tree-subtrees ntree)
                   (tree-subtrees vtree)
                   ))))))

(define (rename exp-tree env-tree)
  (let ((exp (tree-datum exp-tree))
        (env (tree-datum env-tree)))
    (tree (case (first exp)
            ((variable)
             (list 'variable (env-lookup (second exp) env)))
            ((lambda)
             (list 'lambda (env-lookup (second exp) env)))
            (else exp))
          (map rename
               (tree-subtrees exp-tree)
               (tree-subtrees env-tree)))))
```

Figure 3.7: Subroutines for the alpha-renamer, part III.

The final component of the decomposition is the `rename` procedure (Figure 3.7). It takes a node tree and an environment tree and returns a new expression tree in which all lambda-bound variables and variable references are renamed according to the corresponding environment. In the case of our running example:

```
(let ((exp-tree (splay-tree a-term)))
  (let ((defs (filter-formals exp-tree)))
    (rename exp-tree
            (environment defs
                         (fresh-names defs)))))
```

$\stackrel{\text{eval}}{\Longrightarrow}$

```
((lambda x_0)
 ((lambda y_1)
  ((call)
   ((lambda x_2)
    ((variable x_2)))
   ((lambda y_3)
    ((variable x_0))))))
```

Putting all the components together yields the complete alpha renamer, which works as advertised on the test term:

```
(alpha-rename a-term)
```

$\stackrel{\text{eval}}{\Longrightarrow}$

```
(lambda x_0
  (lambda y_1
    (call
     (lambda x_2
       x_2)
     (lambda y_3
       x_0))))
```

The signal processing organization for the alpha renaming program has numerous modularity advantages over the monolithic approaches:

- The tree-manipulation procedures can be designed, implemented, and debugged independently. In the monolithic versions, it was impossible to perform any of these tasks on only one of the subprocesses since they all were intertwined.

- Many program modifications can be made by local changes to the tree-manipulation procedures. For example, in order to make a name generator that walks right-to-left rather than left-to-right, it is only necessary to replace the lr-pre-number within `fresh-names` by an appropriately defined `rl-pre-number`. Such a modification is entirely local; no other module need be changed.

- The tree-manipulation procedures are reusable. Structurally, they share a standard interface (trees) that makes them easy to mix and match. More importantly, the modules can be designed to make minimal assumptions about the contexts in which they are to be used, thereby broadening their range of applicability. For example, the `environment` procedure embodies a downward flow of binding information for *any* tree of names and *any* tree of values, as long as they have the same shape. More context-dependent modules, such as `filter-formals` and `rename` (both of which contain references to lambda-calculus specific details like the `lambda` keyword) can be generalized to make them more widely applicable.

- Many of the tree-manipulation procedures share a common structure that can be captured by higher-order tree-manipulation procedures. For example, `filter-formals` and `rename` are both instances of a more general tree-mapping process in which each datum of a result node only depends on data of the argument nodes. `Environment`, `glom-tree`, and `lr-pre-number` are instances, respectively, of more general processes that accumulate information down a tree, up a tree, and threaded through a tree. We will consider such generalizations in depth in Chapter 6.

- The modules can capture default behavior in a way that facilitates extensions. Extending the alpha renamer to deal with an `if` construct only requires adding new handlers to the conversion routines `splay-tree` and `glom-tree`:

```
(define (splay-tree exp)
  (cond
   ⋮
   ((if? exp)
    (tree (list 'if)
          (list (splay-tree (test exp))
                (splay-tree (then exp))
                (splay-tree (else exp)))))
   ⋮
   ))
```

```
(define (glom-tree tr)
  (let ((exp (tree-datum tr))
        (subexps (tree-subtrees tr)))
    (case (first exp)
     ⋮
      ((if) (make-if (glom-tree (first subexps))
                     (glom-tree (second subexps))
                     (glom-tree (third subexps))))
      ⋮
      )))
```

In particular, the core modules of the alpha renamer — name generation, environment, renaming — need not be changed at all to handle the `if` construct! The reason is that these modules all have appropriate defaults built in. For example, at any node where a name is not supplied, `environment` simply passes the current environment down to all the subexpressions. This default captures essential behavior of "environmentness" that need be specified in only one place.

To be fair, it is worth noting that adding a new name declaration construct would not be as easy as adding `if`. The problem is that the notions of name declaration and scope are too closely tied to details of the lambda calculus within `filter-formals` and `rename`. However it is possible to generalize these modules so that even new binding constructs are easy to add to the base language.

3.1.4 Some Drawbacks

Unfortunately, the aggregate data approach suffers from some important drawbacks that detract from the benefits of modularity. Chief among these is the overhead of manipulating the aggregate data. Consider `mean-age`. The monolithic version of `mean-age` creates no aggregate data structures, but the list-based version of `mean-age` creates five intermediate lists during its execution. There is a time overhead associated with building these lists and taking them apart. Even more debilitating is the space overhead used to compute and store these lists. Under the standard Scheme evaluation model, memory must contain a list the size of the entire database whenever `map-age`, `map-one`, and `down-accumulate-sum` are applied. Furthermore, whenever the base case is reached in any of the computations specified in Figure 3.1, the size of the implicit control stack must be on the order of the database size as well. These problems plague any program representing cables by lists or trees.

The time overhead is very annoying but not devastating. It increases the running time of the monolithic `mean-age` by a constant multiplicative factor. In many cases it is reasonable to pay this price in order to reap the benefits of modularity. For example, because the modular version makes better use of the programmer's time (since it is easier to write, modify, and debug), it may actually be more cost-effective than a faster program that is harder to write.

The space overhead can be a more serious problem. Whereas the time requirements differed in a constant factor, the space requirements can differ in order of growth. The monolithic version of `mean-age` executes in constant space, while the list-based version requires both stack and heap space linear in the size of the database. A sufficiently large database can exhaust available memory in the list-based version, causing the program to fail. This is an unreasonable price to pay for modularity. Even though standardly available computer memories will continue to grow larger at a rapid pace, it is likely that the standard size of information chunks will grow at an even faster pace. The storage pitfalls of the aggregate data approach will become more problematic over time, not less so.

Furthermore, the straightforward aggregate data technique fails totally in cases where the aggregate structures are conceptually infinite. Infinite data structures can be a powerful way to modularize programs (see [ASS85]). An excellent example [Hug90] is decomposing a game program into a part that generates a game tree, and a part that examines the game tree. This supports modularity because the game tree generator can be designed as an independent unit without regard to the particular ways in which it will examined. But since game trees are typically infinite, this decomposition can stretch the above storage overhead problem beyond the capabilities of any finite memory.

Finally, although time and space overheads are the most important problems with the aggregate data approach, they aren't the only ones. Sometimes it is desirable to control the scheduling of operations from different slivers. For example, if two different slivers use I/O operations, we might want these interleaved in a certain fashion, or we might want all the operations in one sliver to happen before those in another. The aggregate data language is not powerful enough by itself to express such constraints. More generally, there are questions of how to combine the computational shapes of individual idioms to yield a desired shape for the composite. It is often possible to manually interweave several linear iterations into a single linear recursive computations, but the standard aggregate data approach does not allow us to talk in these terms.

3.1.5 Partial Solutions

There are a number of techniques for ameliorating the problems outlined above in special cases, but none is satisfactory in general.

`Cdr`-bashing

The `cdr`-bashing trick can be used to reduce the control space associated with list manipulations. Replacing every instance of `GLOM-LIST` with `CDR-BASHING-LIST-COLLECT` would remove the need for implicit stacks in all of the database examples. A similar trick, in conjunction with pointer reversal, could remove implicit stacks from the tree problems as well. But these tricks ultimately convert implicit stacks to explicit ones. They do not address the more fundemental problem of the space taken by the intermediate aggregate structures.

Lazy Data Structures

Lazy data structures are often suggested as a solution to the space problems described above, but this technique does not work in all cases.[1] A data structure is lazy when the computation of each of its components is delayed until it is required (if ever). In SPS programs based on aggregate data, lazy data can reduce space overhead and permit infinite data structures by effectively allowing a limited kind of coroutining between the slivers.

In some SPS programs, lazy data can eliminate the order-of-growth space overhead associated with the intermediate aggregate structures. For example, consider the following

[1] Lazy data structures are not to be confused with the more general strategy of *lazy evaluation*. Lazy evaluation introduces many space problems of its own (such as the *dragging tail problem* [Pey87]) that will not be detailed here.

procedure for counting the number of employees who earn more than a given amount:

```
(define (fat-cat-count threshold database)
  (running-sum
   (map-one
    (filter-salary threshold
                   (generate-records database)))))
```

Rather than manipulating lists, suppose that the subroutines manipulate Scheme *streams*, a form of lazy lists described in [ASS85]. A stream implementation of these subroutines appears in Figure 3.8.

```
(define (generate-records_stream database)
  (define (gen record)
    (if (end-of-database? record)
        the-empty-stream
        (cons-stream record (gen (next-record record)))))
  (gen (first-record database)))

(define (filter-salary_stream threshold records)
  (cond ((empty-stream? records) the-empty-stream)
        ((> (record-get (head records) 'salary) threshold)
         (cons-stream (head records)
                      (filter-salary_stream threshold (tail records))))
        (else (filter-salary_stream threshold (tail records)))))

(define (map-one_stream str)
  (if (empty-stream? str)
      the-empty-stream
      (cons-stream 1 (map-one_stream (tail str)))))

(define running-sum_stream
  (let ()
    (define (accum str sum)
      (if (empty-stream? str)
          sum
          (accum (tail str) (+ (head str) sum))))
    (lambda (stream) (accum stream 0))))
```

Figure 3.8: A stream-based implementation of the parts used by `fat-cat-count`.

The stream-based version of `fat-cat-count` executes in constant control and data space. This is a remarkable result: the space behavior of a monolithic version of `fat-cat-count` is achieved in a modular aggregate data program without any explicit concurrency or side effects. The operations of the subroutines are interleaved as a result of the delayed evaluation of the second argument of `cons-stream`. This interleaving enables the garbage collector to reclaim the intermediate storage allocated for each record by the time the next one is processed.[2] The side effects necessary for the `cdr`-bashing trick aren't needed here because they are effectively hidden by the memoization employed by streams to avoid the recomputation of delayed values [ASS85].

Unfortunately, lazy data is not a silver bullet. The subtle interactions between laziness and garbage collection can make it hard to predict the storage requirements of a program. As one example of the subtlety, compare the `running-sum`$_{stream}$ Figure 3.8 with the more straightforward version below:

```
(define (running-sum_stream stream)
  (define (accum str sum)
    (if (empty-stream? str)
        sum
        (accum (tail str) (+ (head str) sum))))
  (accum stream 0))
```

If the more straightforward version of `running-sum`$_{stream}$ is used in `fat-cat-count`, then constant space behavior cannot be guaranteed![3]

Even worse, lazy data doesn't always interact well with fan-out. Consider a stream implementation of `mean-age`, which exhibits fan-out by using the records generated from the database in two places:

```
(define (mean-age_stream database)
  (let ((records (generate-records_stream database)))
    (/ (running-sum_stream (map-age_stream records))
       (running-sum_stream (map-one_stream records)))))
```

Because Scheme evaluates procedure arguments in some sequential order, there is a point in the computation when one of the arguments to `/` has been completely evaluated but evaluation of the other has not yet begun. Due to the memoization of stream elements,

[2] The garbage collector may not actually reclaim the intermediate storage after every record is processed. But the fact that the storage can be reclaimed at any future point means that it not charged to the computation in space analysis.

[3] Here's why: Under the usual environment model of Scheme evaluation (see [ASS85]), the `stream` argument to `running-sum`$_{stream}$ is accessible from the environment in which `accum` is evaluated. Even though `stream` is never referenced within `accum`, its value can't be garbage collected until `accum` returns. But in the case of `fat-cat-count`, `stream` will be holding onto the stream created by `map-one`$_{stream}$, whose size is the number of fat cats in the database. So `fat-cat-count` is no longer constant space.

There are implementation strategies and alternate evaluation models under which `accum` won't unnecessarily hang onto `stream`. In these cases, `fat-cat-count` would still be constant space. However, the language doesn't *guarantee* this behavior.

memory must contain an intermediate data structure corresponding to the entire database at this point. If streams were not memoized, then there would be no storage overhead, but the database would have to be traversed twice. This latter case is equivalent to manually removing the fan-out from `mean-age`:

```
(define (mean-age_{no-fan-out} database)
  (/ (running-sum (map-age (generate-records database)))
     (running-sum (map-one (generate-records database)))))
```

In either case, lazy data does not help.

Hughes's Approach

Hughes improved the lazy data approach by supplying mechanisms that can guarantee desirable space behavior even for networks exhibiting fan-out [Hug83, Hug84]. He observed that in programs like `mean-age`, a constant space computation can only be achieved if the arguments to `/` are somehow computed together in lock step. That is, `generate-records` should not produce a new record until all the mappers and summers have completely processed the previous one; this way, no handle on the previous record needs to be stored.

Hughes showed that in functional languages with sequential argument evaluation it is *impossible* to express this kind of lock step evaluation in a modular way. To overcome the limitations of sequential argument evaluation, Hughes introduced mechanisms for parallel evaluation (`par`) and synchronization (`synch`). These are summarized in Figure 3.9.[4]

Using Hughes's `par` and `synch` constructs, the stream-based version of `mean-age` can be forced to exhibit constant space behavior as follows:

[4] Hughes originally posed his mechanism in a lazy functional language, so these versions have to be suitably modified to make sense in Scheme. Hughes's (`synch` E) actually returned a pair both of whose components held the value of E; but each component could only be accessed when *both* components had been requested.

- `(par` E`)` returns immediately with a placeholder for the result of the evaluation of E. The evaluation of E proceeeds concurrently with the rest of the computation. Any context requiring the actual value of the placeholder will wait until the value is available. `Par` is equivalent to the `future` construct provided by many Lisps [Hal85, Mil87, For91].
- `(synch` E`)` returns an object that suspends the computation of E. `Demand1` and `demand2` are procedures that demand such an object to return the value of its suspended computation, but the computation is only initiated when *both* `demand1` and `demand2` have been called on the object. Once computed, the value of E is memoized. `Synch` is similar to Scheme's `delay`, except that there are two different forcing functions, both of which must be invoked before the delayed expression is computed.

Figure 3.9: Scheme renditions of Hughes's `par` and `synch` constructs.

```
(define (mean-age_hughes database)
  (let ((records (synch-stream (generate-records_stream database))))
    (/ (par (running-sum_stream
              (map-age_stream
                (map-stream demand1 records))))
       (par (running-sum_stream
              (map-one_stream
                (map-stream demand2 records)))))))

(define (synch-stream str)
  (if (empty-stream? str)
      (synch the-empty-stream)
      (cons-stream (synch (head str))
                   (synch-stream (tail str)))))

(define (map-stream f str)
  (if (empty-stream? str)
      the-empty-stream
      (cons-stream (f (head str))
                   (map-stream (tail str)))))
```

`Mean-age`$_{hughes}$ uses `synch-stream` to associate a synchronization point with every database record and `par` to evaluate the operands of `/` concurrently. Since the age computation uses `demand1` to access a record and the length computation uses `demand2`, neither computation can race ahead of the other. It's as if both computations must pass through a series of locked doors, each of which has two locks, and each computation has they key to only one of the locks. As in the case of `fat-cat-count`, constant space behavior is once again guaranteed.

Hughes's technique is a good mechanism for controlling space behavior, but it has a few drawbacks:

- *Lack of abstraction*: The technique requires extending the sliver diagram for `mean-age` with two instances of `par` and three slivers that manage synchronization (`synch-stream` and two instances of `map-stream`). It would be preferable to somehow abstract over these parallelization and synchronization operations so that the original network structure could be maintained. Unfortunately, they are difficult to capture in a reusable fashion. For example, `synch-stream` can't simply be inserted into the definition of `generate-records`$_{stream}$ because other applications of this procedure might exhibit a fan-out other than two. Without extra support from the language, programmers are required to manage `par` and `synch-stream` explicitly — an error-prone prospect, especially since since `synch` raises the specter of deadlock.[5]

- *Weak synchronization*: Using `synch-stream` to replace every instance of fan-out in a sliver diagram does not force all the slivers in the diagram to compute in lock step. Since synchronization is local, not global, there may be a lag between two sliver computations that's related to the number of `synch-stream`s that appear between them. This is usually not very important, but can be troublesome if tighter operator interleaving is desired.

The concurrency and synchronization mechanisms underlying my technique are closely related to Hughes's `par` and `synch` except that they address the two drawbacks above. In particular, my technique makes it possible to hide the mechanisms inside of sliver abstractions so that the programmer does not have to deal with them.

Program Transformations and Compilation Techniques

There are a large number of program transformations and compilation techniques that can eliminate some intermediate data structures from aggregate data programs by fusing connected slivers. They are all essentially high-level versions of the classic low-level *loop*

[5] *Deadlock* is a state in which a computation can make no progress. Deadlock can arise in the presence of `synch` if `demand2` cannot be applied until the successful return of `demand1`, or vice versa.

fusion technique performed by many optimizing compilers [ASU86]. As a typical example of these techniques, the Scheme expression

```
(map f (map g l))
```

could be replaced by

```
(map (lambda (x) (f (g x))) l).
```

The later expression is preferable because no list is constructed for the ouput of `g`.

The problem with existing transformation and compilation techniques is that they either provide no guarantees or they only work on a limited class of programs. Algebraic transformation techniques [DR76, Dar82, Bac78, Bel86, Bir89a, Bir86, Bir88] perform transformations like the above `map` removal, but systems that automatically apply these transformations do not guarantee that all intermediate data will be removed. Because programmers cannot depend on the transformations, they must seek other methods of controlling the space behavior of their programs. APL compilation techniques [GW78, Bud88] suffer from the same problem.

The listlessness [Wad84, Wad85] and deforestation [Wad88, Chi92, GLJ93] techniques pioneered by Wadler *do* provide guarantees, but they are rather limited in applicability. Listlessness handles a subclass of list programs, but no trees. Deforestation can eliminate both lists and trees, but only in networks that exhibit no fan-out.

The most impressive of the transformation approaches is Waters's *series* technique [Wat91, Wat90]. Series is a linear datatype that corresponds to a sequence of the values assumed by a state variable during an iteration. The series compiler can transform a large class of series networks, including those with fan-in, fan-out, and filtering, into efficient loops. The most important aspect of series is that there is an explicit set of easy-to-check restrictions that the programmer can verify to determine whether a given network can be efficiently compiled. The compiler generates a warning when these restrictions are violated.

Alas, series is limited to the expression of linear iterative computations. It cannot handle tree-shaped computations, or even linear recursions. Cyclic data dependencies are not allowed, even though these are sometimes useful for program decomposition.[6] Finally, because it is based on static analysis, series requires that the network of series operators be determinable at compile-time. This limits expressiveness by outlawing the dynamic configuration of the network at run-time. The lock step processing technique developed in Chapter 5 is the basis for dynamic version of series that addresses all of these issues.

The problem with all of the transformations described in this section is that they are too restrictive (e.g., can't handle tree-shaped data, fan-out, or recursion) and/or they fail to provide provide the programmer with sufficient guarantees about improvements. The upshot is that programmers often shun the aggregate data approach and instead embrace the fine-grained control of the monolithic style.

The storage overhead problem is a classic example of how modularity can preclude a programmer from expressing desired behavior. The aggregate data approach hides the operational details of the slivers inside of procedures specified wholly in terms of their input/output behavior. The programmer has no hook into how the procedures work, and therefore cannot express details like lock step evaluation.

3.2 The Channel Approach

In the channel approach, SPS networks are viewed as interconnected processes that communicate via data channel defined by the cables. Whereas the focus in the aggregate data approach is the data transmitted on a cable, the focus in the channel approach is the process that sends and receives data from the channel. A characteristic of the channel approach is that the collection of elements transmitted across a channel is not treated as a single entity. In the channel approach, processes are usually assumed to be independent threads of control. They may be executing concurrently, or they may be coroutining in some fashion.

The channel approach is supported by numerous languages and systems. Hoare's CSP is the canonical version of this approach [Hoa85]; Unix pipes [KP84] is one of the most widely used. Other examples of the channel approach include: communicating threads

[6] Waters argues that such cycles make programs harder to understand, and therefore should be avoided at all costs.

[Bir89b, CM90], producer/consumer models (CLU iterators [L^+79], Id's I-structures and M-structures [ANP89, Bar92], Linda [CG89]) and dataflow ([Den75], [WA85], [DK82]).

Below, we explore both coroutining and concurrent versions of the channel approach. We will see that the channel approach can provide reasonable control over space behavior but that it is not a very good approach for expressing linear recursions and tree shaped computations.

3.2.1 Coroutining Example

As a demonstration of coroutining, consider a simple organization in which each sliver is represented as an state-maintaining object that keeps track of the source object for each of its input cables (this is the representation of the "channel"). Computation will be performed in a demand-driven fashion. If an object receives a request for the next value on one of its output cables, it may in turn request values from its input cables in order to satisfy the request.[7]

Figures 3.10 and 3.11 present encodings of the database idioms under this organization. Every object (except for record generators) has a `source` variable that indicates the source of its input wire. Each object is represented as a thunk (parameterless procedure) that returns its next output every time it is called. The `demand!` procedure enhances the readability of this usage pattern:

```
(define (demand! object) (object))
```

When an object runs out of values to produce, it returns a distinguished `done` value:[8]

```
(define done '(*done*))
(define (done? obj) (eq? obj done))
```

A detail: the `set!`s used within the accumulators guarantee that they will return `done` after producing the accumulated value.

Here's a version of the `fat-cats` program in this approach:

[7] In more complex organizations, objects might also keep track of the targets of their output cables, and computation might exhibit both data-driven and demand-driven aspects.

[8] An alternate approach is for every object to handle both an "are you done?" message and a "give me your next value" message. It turns out that this alternate approach unduly complicates the filtering idiom in this style (try it and see!).

```
(define (make-record-generator_cor database)
  (let ((record (first-record database)))
    (lambda ()
      (if (end-of-database? record)
          done
          (let ((val record))
            (set! record (next-record record))
            val)))))

(define (make-running-summer_cor source)
  (define (accum sum)
    (let ((next (demand! source)))
      (if (done? next)
          (begin (set! accum (lambda (x) done))
                 sum)
          (accum (+ next sum)))))
  (lambda () (accum 0)))

(define (make-stacking-list-collector_cor source)
  (define (accum)
    (let ((next (demand! source)))
      (if (done? next)
          (begin (set! accum (lambda () done))
                 '())
          (cons next (accum)))))
  (lambda () (accum)))
```

Figure 3.10: Generating and accumulating idioms for the database example in the coroutining technique.

```
(define (make-mapper-maker_cor fun)
  (lambda (source)
    (lambda ()
      (let ((next (demand! source)))
        (if (done? next)
            done
            (fun next))))))

(define make-age-mapper_cor
  (make-mapper-maker_cor (lambda (r) (record-get r 'age))))

(define make-name-mapper_cor
  (make-mapper-maker_cor (lambda (r) (record-get r 'name))))

(define make-1-mapper_cor
  (make-mapper-maker_cor (lambda (r) 1)))

(define (make-salary-filter_cor threshold source)
  (define (next)
    (let ((record (demand! source)))
      (cond ((done? record) done)
            ((> (record-get record 'salary)
                threshold)
             record)
            (else (next)))))
  (lambda () (next)))
```

Figure 3.11: Filtering and mapping idioms fort the database example in the coroutining technique.

```
(define (fat-cats_cor threshold database)
  (demand! (make-stacking-list-collector_cor
            (make-name-mapper_cor
             (make-salary-filter_cor threshold
                                      (make-record-generator_cor database))))))
```

This resembles the aggregate data approach in the way that the interconnecton of slivers is specified by nesting expressions. However, no intermediate aggregate structures are created. Instead, demand and values pass back and forth between the objects in a coroutining fashion. This computational pattern closely resembles the evaluation of a lazy data version of `fat-cats`.

The `mean-age` program is harder to express in this approach due to fan-out. The naive approach, shown below, fails to work as desired:

```
;;; Doesn't work because of fan-out!
(define (mean-age_cor-wrong database)
  (let ((generator (make-record-generator_cor database)))
    (/ (demand! (make-running-summmer_cor
                 (make-age-mapper/dist generator)))
       (demand! (make-running-summmer_cor
                 (make-1-mapper/dist generator))))))
```

Because requesting a record from `generator` changes its state, the two mappers connected to `generator` receive different record sequences!

To avoid this problem without traversing the database twice, it is necessary to buffer values produced by the generator until both mappers have consumed them. This can be done in general by the using the $\texttt{copy}_{cor}$ routine shown in Figure 3.12 to represent cable fan-out in a sliver diagram. $\texttt{Copy}_{cor}$ uses a buffer to effectively return two copies of a single object. Using this routine, `mean-age` can be correctly implemented as:

```
(define (mean-age_cor database)
  (mlet (((gen1 gen2) (copy_cor (make-record-generator/dist database))))
    (/ (demand! (make-running-summer_cor
                 (make-age-mapper_cor gen1)))
       (demand! (make-running-summer_cor
                 (make-1-mapper_cor gen2))))))
```

Introducing buffering makes it possible to handle fan-out, but it brings with it the same space overhead problems that hamper the aggregate data approach. In a language with sequential argument evaluation, the $\texttt{mean-age}_{cor}$ procedure will end up buffering the entire database at some point. As in the aggregate data approach, the problem with the

```
(define (copy_cor source)
  (let ((queue (make-queue))
        (slowpoke 0)) ; 0 = neither copy behind;
                      ; 1 = first copy behind;
                      ; 2 = second copy behind.
    (define (make-copy my-id other-id)
      (lambda ()
        (if (= my-id slowpoke)
            ;; If I'm behind, get next value from queue.
            (let ((head (dequeue! queue)))
              (if (queue-empty? queue)
                  (set! slowpoke 0))
              head)
            ;; If I'm not behind, get next value from source and queue it.
            (let ((next (demand! source)))
              (begin
                (enqueue! next queue)
                (set! slowpoke other-id)
                next)))))
    ;; Return a list of two thunks that act like SOURCE.
    (list (make-copy 1 2) (make-copy 2 1))))
```

Figure 3.12: A routine that returns two copies of a given object. The procedures `make-queue`, `enqueue!`, `dequeue!`, and `queue-empty?` implement a queue abstraction that is not detailed here.

coroutining technique is that the modules preclude the programmer from expressing fine-grained control. In both cases, some form of concurrency and synchronization are required to express desired control.

A new problem exhibited by coroutining is that tree-shaped computations are hard to handle. Using demands to request the next value from an input cable is fine when the values are arranged linearly in sequences, but is problematic when they are arranged as trees. How would `alpha-rename` be coded in the coroutining technique? There are several approaches, none of which is entirely satisfactory:

- Encode the tree as a linear sequence of its contents along with extra information indicating the shape of the tree. This kind of encoding is used all the time to transmit structured information over physical wires. Yet, a programmer doesn't want to have to think in these terms; at the very least, there need to be abstractions for the encoding and decoding process.

- Associate with each request an address that indicates the desired element. This complicates both requestor and requestee by requiring them to keep track of address information.

- Have each object respond to `demand!` by returning subtree objects in addition to the usual value. The requester then can explicitly choose which subobject to `demand!` next. This complicates each requester by forcing it to manage the subobjects.

3.2.2 Concurrent Process Example

We address the concurrency problem of the coroutining technique by considering a model in which the sliver processes are concurrent. The processes need not actually be running in parallel on separate physical processors; for our purposes, simulated concurrency on a single processor is perfectly adequate.

The Database Example

Versions of `mean-age` and `fat-cats` written in the concurrent process technique are shown in Figure 3.13. They make use of the idioms presented in Figures 3.14 and 3.15. In this code, we assume that Scheme has been extended with some constructs that support concurrency:[9]

- `(cobegin` *exp1 exp2* ...`)` evaluates each of the subexpressions in its own concurrent thread. Evaluation of the subexpressions may be arbitrarily interleaved. `Cobegin` returns only when all component thunks have returned; its return value is a list of the subexpression values.

- `(make-channel)` returns a new channel object, which acts as a FIFO communication queue between concurrently executing threads.

- `(send!` *channel value*`)` tacks *value* on the end of the *channel* queue and returns *value*.

- `(receive!` *channel*`)` removes the first value from the *channel* queue and returns it. If the queue is empty, then `receive!` blocks until a value is available to read.

Channels are assumed to be buffered communication queues in which sends and receives are performed by side effect, but there are many other communication models that could have been chosen. As in the coroutining technique, `done` and `done?` are used to indicate the termination of a value sequence.

In `mean-age`$_{conc}$ and `fat-cats`$_{conc}$, each cable is represented explicitly as a channel object, and each sliver is represented as a channel-manipulation procedure. The `cobegin` allows the slivers conceptually to execute in parallel, though on a sequential machine their computations are actually interleaved. Most slivers wait for values from their input channels, and then produce values on their output channels. The blocking behavior of `receive!` on an empty channel is a synchronization mechanism that forces slivers to wait for values to become available on their input channels before proceeding. This results in a data-driven model of evaluation that stands in contrast with the demand-driven evaluation of the lazy data technique and the coroutining technique.

The rationale behind the `copy`$_{conc}$ is the same as that for `copy`$_{cor}$ in the previous example. Since `receive!` works by side effect, two consumers sharing the same channel will not see the same sequence of values. Copying the values to two separate channels decouples the

[9] Except for a few cosmetic changes changes, these constructs are the ones described in [JG89].

```
(define (mean-age_conc database)
  (let ((gen->copy      (make-channel))
        (copy->map-age  (make-channel))
        (copy->map-one    (make-channel))
        (map-age->sum1  (make-channel))
        (map-one->sum2    (make-channel)))
    (mlet (((_ _ _ _ sum1 sum2)
            (cobegin
              (generate-records_conc database gen->copy)
              (copy_conc gen->copy copy->map-age copy->map-one)
              (map-age_conc copy->map-age map-age->sum1)
              (map-one_conc copy->map-one map-one->sum2)
              (running-sum_conc map-age->sum1)
              (running-sum_conc map-one->sum2))))
      (/ sum1 sum2))))

(define (fat-cats_conc threshold database)
  (let ((gen->filter    (make-channel))
        (filter->map    (make-channel))
        (map->list      (make-channel)))
    (mlet (((_ _ _ name-list)
            (cobegin
              (generate-records_conc database gen->filter)
              (filter-salary_conc threshold gen->filter filter->map)
              (map-name_conc filter->map map->list)
              (stacking-list-collect_conc map->list))))
      name-list)))
```

Figure 3.13: Versions of `mean-age` and `fat-cats` in the concurrent process technique. The `mlet` construct is the pattern matching version of `let` introduced earlier; the underscore character '_' is used in patterns to indicate unnamed slots.

```
(define (generate-records_conc database out)
  (define (gen record)
    (if (end-of-database? record)
        (send! out done)
        (begin
          (send! out record)
          (gen (next-record record)))))
  (gen (first-record database)))

(define (running-sum_conc in)
  (define (sum ans)
    (let ((next (receive! in)))
      (if (done? next)
          ans
          (sum (+ next ans)))))
  (sum 0))

(define (stacking-list-collect_conc in)
  (define (accum)
    (let ((next (receive! in)))
      (if (done? next)
          '()
          (cons next (accum)))))
  (accum))
```

Figure 3.14: Generating and accumulating idioms for the database example in the concurrent process approach.

```
(define (make-map_conc f)
  (define (map in out)
    (define (loop)
      (let ((next (receive! in)))
        (if (done? next)
            (send! out done)
            (begin
              (send! out (f next))
              (loop)))))
    (loop))
  map)

(define map-age_conc  (make-map_conc (lambda (r) (record-get r 'age))))
(define map-name_conc (make-map_conc (lambda (r) (record-get r 'name))))
(define map-one_conc  (make-map_conc (lambda (x) 1)))

(define (filter-salary_conc threshold in out)
  (define (loop)
    (let ((next (receive! in)))
      (if (done? next)
          (send! out done)
          (begin
            (if (> (record-get next 'salary)
                   threshold)
                (send! out next))
            (loop)))))
  (loop))

(define (copy_conc in out1 out2)
  (define (loop)
    (let ((next (receive! in)))
      (begin
        (send! out1 next)
        (send! out2 next)
        (if (not (done? next)) (loop)))))
  (loop))
```

Figure 3.15: Mapping, filtering, and copying idioms for the database example in the concurrent process approach.

consumers from each other. An alternate approach would be to share a single channel but provide each consumer with its own pointer into the channel queue.

Storage Overhead

The concurrent process technique offers the possibility of better storage behavior than the other approaches, but does not guarantee it. Consider `mean-age`$_{conc}$. If all of the component processes are operating in lock step, then no channel queue ever contains more than one element and the program as a whole uses only constant space. However, at the other extreme, the `map-age`$_{conc}$ process might send all database ages to its output channel before the `running-sum`$_{conc}$ at the other end of the channel receives the first one. In this case, the `map-age->sum1` channel requires storage linear in the size of the database.

Channel storage requirements can be reduced by constraining the unbounded nature of channel queues. The problem with unbounded channels is that they allow a producer to race arbitrarily far ahead of one of its consumers. A common solution is to limit the number of values that the channel can buffer.

In the most restrictive approach, channels aren't allowed to buffer any values; instead, the processes at the ends of the channel engage in a *rendezvous* in which a `send!` does not return in the sending process until a `receive!` has been executed in the receiving process. This is the approach adopted by CSP [Hoa85]. A rendezvous-based version of `mean-age`$_{conc}$ would be guaranteed to require only constant space.

A less restrictive approach is a *bounded queue*, in which there is an upper limit on the size of a queue; when the queue reaches this limit, a `send!` blocks until the queue gets smaller. `mean-age`$_{conc}$ would also use only constant space under this approach, though the constant would be larger than in the rendezvous case. The bounded queue approach is a standard means of implementing flow control between independent processes [Bir89b].

Why not simply adopt a policy in which all the channel queues are bounded? The problem with this is that there are problems that require unbounded queues. Consider an `append`$_{conc}$ process with two input channels and one output channel that first copies all elements from the first input channel to the output channel and then copies all elements from the second input channel to the output channel (Figure 3.16).

```
(define (append_conc in1 in2 out)
  (define (copy1)
    (let ((next (receive! in1)))
      (if (done? next)
          (copy2)
          (begin (send! out next)
                 (copy1)))))
  (define (copy2)
    (let ((next (receive! in1)))
      (if (done? next)
          (send! out next)
          (begin (send! out next)
                 (copy2)))))
  (copy1))
```

Figure 3.16: A procedure that sends to its output channel the result of appends the values from its two input channels.

If `append`$_{conc}$ is used in the following `twice` procedure,

```
(define (twice in out)
  (mlet (((c1 c2) (copy_conc in)))
    (append_conc c1 c2 out)))
```

then the second input channel `c2` to `append`$_{conc}$ requires a queue whose size is the number of elements in the `in` channel. Since this size may be arbitrarily large, bounded queues do not suffice in general.

A reasonable compromise is to let the programmer choose between bounded and unbounded channels. In addition to the unbounded `(make-channel)`, the language could also provide a `(make-bounded-channel` *num*`)` constructor that creates a channel whose queue length is bounded by *num*. Then for cases like `mean-age`, the programmer has a means of expressing the constant-space nature of the computation while maintaining the modular SPS structure. Its ability to support both modular decomposition and some means of constraining control distinguishes the concurrent process technique from the other techniques we have studied.

Deadlock

While the concurrent process technique has some nice properties, there are a number of issues that detract from it. Foremost among these is that concurrent programs can sometimes enter a wedged state called *deadlock* in which processes aren't done executing, but none can make any progress. Here is a simple example of a program that deadlocks:

```
(let ((c1 (make-channel))
      (c2 (make-channel)))
  (cobegin
    (begin (receive! c2) (send c1 19))
    (begin (receive! c1) (send c2 23))))
```

Each of the two processes created by the `cobegin` attempts to receive a value from the other process before it sends a value to the other process. Since neither process can make headway, the program is stuck and no value is returned.

The problem with the above example is that the sequential ordering constraints implied by the `begin`s are too strong and cannot be satisfied. Deadlock can often be avoided by removing spurious ordering constraints. For instance, the above example can be made to work by changing the `begin`s to `cobegin`s:

```
(let ((c1 (make-channel))
      (c2 (make-channel)))
  (cobegin
    (mlet (((_ ans1)
            (cobegin (receive! c2) (send! c1 19))))
      ans1)
    (mlet (((_ ans2)
            (cobegin (receive! c1) (send! c2 23))))
      ans2)))
```
$\stackrel{\text{eval}}{\Longrightarrow}$ `(19 23)`

This change is reasonable because the `send!`s do not depend on the values of the `receive!`s. However, when there is an inherent dependency loop, deadlock is unavoidable and indicates a program bug. Here is a deadlocking expression containing such a loop:

```
(let ((c1 (make-channel))
      (c2 (make-channel)))
  (cobegin
    (send c1 (receive! c2))
    (send c2 (receive! c1))))
```

The above examples are contrived, but spurious deadlocks are easy to come by in practice. Avoiding them requires a defensive programming style that relaxes unnecessary ordering constraints. For instance, the unnecessary ordering of `send!`s in the $\texttt{copy}_{conc}$ procedure from Figure 3.15 can insidiously lead to deadlock in some contexts. This procedure can be written more robustly by decoupling the `send!`s with a `cobegin`:

```
(define (copy_{conc-better} in out1 out2)
  (define (loop)
    (let ((next (receive! in)))
      (begin
        (cobegin               ; Relax ordering of SEND!s
          (send! out1 next)
          (send! out2 next))
        (if (not (done? next)) (loop)))))
  (loop))
```

In the presence of bounded channels, deadlock is even more of a threat because it can be caused simply by choosing a bound that is lower than actually required.

Deadlock complicates the concurrent processing approach to SPS by requiring the programmer to reason more carefully about sliver composition than in other styles. In the other approaches, the meaning of a network is easy to determine from the meaning of the parts; as long as there are no *directed* loops between slivers, there are no surprises. However, in the concurrent processing style, deadlock can arise from *undirected* loops[10] or channels whose buffers aren't big enough. The extra reasoning necessary to show that a program does not have deadlock is part of the price of additional control.

Other Issues

Deadlock is not the only issue that complicates SPS programs written in the concurrent processing style. Here are some others:

- *Synchronization:* Deadlock arises from overconstrained control. But underconstrained control also leads to problems, usually involving side effects. Consider the following version of `copy`:

```
(define (copy_{conc-worse} in out1 out2)
  (define (loop)
    (let ((next (receive! in)))
      (cobegin
        (send! out1 next)
        (send! out2 next)
        (if (not (done? next)) (loop))))) ; ***
  (loop))
```

[10] An undirected loop between slivers is formed by any two distinct paths from one to another.

This differs from $\texttt{copy}_{conc-better}$ in that the starred line appears inside the `cobegin`. This allows the next value to be received in parallel with sending the current value. Unfortunately, it also allows the next value to be sent *before* the current value is sent. This changes the input/ouput behavior of `copy` in a major way.

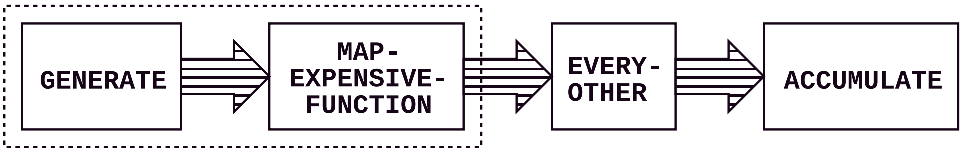
Underconstrained control is a classic problem whenever concurrent processes communicate via shared mutable data. The solution is to introduce extra constraints in the form of synchronization. Typical synchronization mechanisms include locks [Bir89b], semaphores [Dij68], monitors [Hoa74], I-structures [ANP89], M-structures [Bar92], and Hughes's `synch` construct [Hug83, Hug84].

- *Termination:* A `cobegin` does not return until all its subprocesses have returned; this kind of behavior is typical of fork/join parallelism. But getting all the subprocesses to terminate can be tricky. Imagine an accumulator that only consumes the first few values of a long or unbounded sequence produced by a generator. The accumulator returns quickly; but what forces the generator to stop and return? For these kinds of situations, the concurrent process technique often requires a special means of terminating processes.

  An alternate approach is to allow a process to return before it terminates. This decoupling is employeed in so-called *eager* or *lenient* evaluation strategies ([Tra88, Hal85, Mil87]) in which computations may continue to execute after a final answer has been returned. Some means of garbage-collecting these superfluous computations is necessary to avoid wasting resources.

- *Tree-shaped Data:* Like the coroutining technique, the concurrent process technique has difficulty expressing tree-shaped computations. The same circumventions suggested for coroutining work here as well. But a more satisfying approach is to mix aspects of the data aggregate approach and the concurrent process approach. For example, rather than transmitting a tree over a single channel, why not transmit a tree over a tree of channels, where each channel transmits exactly one element? This is the approach taken by languages, such as Id [AN89] and Linda [CG89], that exhibit *producer/consumer parallelism.* These languages support *non-strict data structures* that can be manipulated by a program before their components have been computed. Any attempt to reference an uncomputed component results in a computation that is blocked until the component is actually there. Thus, components of a non-strict data structure are essentially channels that can communicate only a single element before being "used up".

- *Unnecessary Computation:* Data-driven evaluation can often perform unnecessary operations. As a simple example, imagine a network where an expensive mapper precedes a filter that passes only every other element:



  In this case, much of the work being done by the mapper is for naught. This contrasts with demand-driven evaluation, in which only those operations necessary for the final result are performed. An obvious fix is to swap the components, but the mapper may be packaged together with other slivers in a way that makes this infeasible.

  In systems that provide real parallelism (i.e., many physical processors), it is often possible to reduce overall computing resources by wasting or repeating some computation. (The data parallel techniques discussed in [HS86a] are an excellent example of this phenomenon.) In such systems, demand-driven techniques can reduce performance rather than improving it. However, in systems providing only simulated parallelism (i.e., a single processor), demand-driven techniques may still be advantageous. Pingali and Arvind show how demand-driven computation can be simulated within a dataflow model [PA85], [PA86].

- *Overconstrained Operation Order:* The linearity of channels sometimes overconstrains the order in which operations are applied. Consider the mapping routines in Figure 3.15. These routines force the mapped function to be applied to each record in the order in which it appears in the database. Compare this behavior with the mapping functions for the aggregate data approach. In the latter case, the mapped function can happen in any order with respect to the recursive mapping. This unspecified order can increase the modularity of components like mappers, which can then be used in contexts that constrain the order in different ways. It is unwise to specify operation scheduling too early!

3.3 Other Techniques

Here we discuss a few common modularity techniques that are clearly relevant to the expression of loops and recursions, but can't easily be characterized within the aggregate data approach or the channel approach.

3.3.1 Higher Order Procedures

In a programming language that supports higher-order procedures, it is natural to capture general loops and recursions as procedures. For example, here are Scheme procedures encapsulating iteration and linear recursion:

```
(define (iterate initial-state done? down final)
  (define (iter state)
    (if (done? state)
        (final state)
        (iter (down state))))
  (iter initial-state))

(define (recur initial-state done? base down up final)
  (define (rec state)
    (if (done? state)
        (base state)
        (up state (rec (down state)))))
  (final (rec initial-state))))
```

While this is an elegant and powerful strategy for capturing control constructs, it does not provide the kind modularity supported by the signal processing style.[11] For example, implementing `mean-age` via `iterate` requires the programmer to manually interweave the `done?` parts of the component slivers, the `down` parts of the components, and the `final` parts of the components. This is hardly an improvement over the monolithic approach! The

[11] Another problem is that general tree walkers are considerably more complex than the simple linear examples shown here.

problem is that the looping control structure is still centralized in `loop` rather than being distributed over components.

Modularity could be achieved in this approach by representing an iterative sliver as a record of procedures that described the sliver's contribution to each of the arguments of the abstracted control structure. For example, an iterative sliver would be a record of `initial-state`, `done?`, `down`, and `final` components. However, it would also be necessary to define an appropriate means of combination for such records, which would be complicated in general. While this approach may not be reasonable for programmers, it can be good idea for compilers. Indeed, the notion of gluing together corresponding fragments of different slivers is at the heart of Waters's series compiler [Wat91].

3.3.2 Looping Macros

Many versions of Lisp have supported complex macros that capture certain looping idioms.[12] For example, in Common Lisp [Ste90], the `mean-age` procedure can be written as follows:

```
(defun mean-age (database)
  (loop for record = (first-record database)
                 then (first-record (next-record record))
        until (end-of-database? record)
        sum (record-get record 'age) into total
        count record into length
        finally (return (/ total length))))
```

Here, `for`, `until`, `sum`, `count`, and `finally` all introduce clauses into the occurence of the `loop` looping macro. Each clause is intended to express a common iterative processing idiom in a convenient syntax.

As with the higher-order procedure approach, looping macros suffer modularity problems. Again, there is a centralized control structure that is not distributed over the components. There is no way to abstract over a collection of clauses to reuse them in other occurrences of `loop`. While some clauses concisely capture an idiom (e.g., the `sum` and `count` clauses), other idioms are spread over several clauses (e.g., record generation). This approach is essentially an alternate syntax for writing a monolithic iteration.

[12] As far as I know, there are not any similar macros that capture the idioms of tree recursions. Designing such a macro would be an interesting project.

3.3.3 Attribute Grammars

Attribute grammars are a formalism invented by Knuth [Knu68] for declaratively specifying the decoration of tree nodes with named attributes. Though originally intended for describing the semantics of programming languages based on their grammars, today they are mainly used for syntax-directed compilation techniques (see [DJL88] for an overview).

Attribute grammars are related to slivers because (1) they are a language for describing a class of tree computations (2) they support a crude notion of shape. Attribute grammars specify computations on a tree (typically a parse tree) by indicating how an attribute at one node in a tree is calculated from attributes at the current node, its parent node, or its children nodes. Attributes are classified according to dependency information: an attribute computed from parent nodes is said to be *inherited*, while one computed from children nodes is said to be *synthesized*. "Inherited" and "synthesized" roughly correspond to *parallel down* and *parallel up* computation shapes. Indeed, it is possible to view attribute grammars as defining recursive functions [Joh87] or procedures [Kat84].

Classical attribute grammars suffer from a lack of modularity because they distribute the specification of attribute computations across all the different types of tree nodes. However, in recent years, there has been a flurry of activity on *modular attribute grammars*, which strive to group all the computations of a given attribute into a modular unit [DC90, Ada91, FMY92, KW92, Wat92].

While attribute grammars can be a powerful declarative framework for specifying tree computations, they do not seem to be good languages for controlling the behavior of these computations:

- As far as I can tell, most of the attribute grammar frameworks assume that the entire tree being decorated resides in memory. This is an exponential difference in space complexity for monolithic tree-recursive procedures that require space proportional only to the depth of the tree.

- Attribute grammars don't supply a means of expressing tail recursion. To return a result from the leaf of a tree, it is necessary to pass it through all the intermediate nodes back to the root, even if there are no pending computations to be performed on it.

- The declarative nature of attribute grammars can make it difficult to predict how many passes the attribute computation will make over the tree. (There are some frameworks, such as one-pass attribute grammars [Kos84], that do limit the number of passes).

- Attribute grammar formalisms are usually not designed to express general tree computations; they are typically tied to parsing technology and are used mainly to specify language implementations.

- Modular attribute grammar formalisms typically involve ad hoc mechanisms for specifying component connectivity. Connections are much more natural to express in the aggregate data and channel approaches.

3.4 Summary

We have examined in detail issues of modularity and control for a linear and tree-based example. The following themes stand out:

- Modularity arguments suggest that some programs can be structured like signal processing networks of mix and match slivers that embody idiomatic loops and recursions. This approach is called the signal processing style of programming.

- The modularity of slivers is enhanced when their specification leaves open certain details that can be filled in by the context in which they are used. Examples of this theme include:

  - *Higher-order procedures* capture general patterns while allowing parameters to be supplied where they are used.

  - *Lazy data* permits the specification of potentially infinite data structures, but these are only computed to the extent required by the context in which they're used.

  - *Concurrency* allows operations in different slivers to be interleaved, subject to the ordering constraints that result from their connection. The reusability of concurrent processes is increased when spurious ordering constraints on their computations are removed.
  - *Unspecified sequential evaluation of arguments* gives a limited form of operator interleaving.

- *Side effects* are a crucial mechanism both for achieving modularity and specifying fine-grained control. They aid modularity by localizing computation and reducing the interfaces between parts. They permit fine-grained control of space resources and operation order, as in the `cdr`-bashing list collection example.

- Traditional forms of implementing input/output modularity often preclude the programmer from controlling operational details of a program at a fine-grained level. Such details include time and space complexity, operation order, and limiting unnecessary computations.

- Standard techniques for SPS programming have benefits and drawbacks along various dimensions:

  - *Storage requirements* and operation scheduling are difficult to control in the aggregate data and coroutining techniques. They are easier to control in presence of concurrency, but then it is necessary to contend with issues such as deadlock and termination.
  - *Fan-out* in the cables of sliver diagrams causes no problems for the aggregate data style, but requires special handling in the channel techniques. Undirected cycles (formed by fan-out in combination with fan-in) can also lead to deadlock in the concurrent process technique.
  - *Unnecessary computation* can be avoided by using demand-driven evaluation. This form of evaluation is natural in the lazy aggregate data style and the demand-driven coroutining approach. However, it is in conflict with the natural data-driven evaluation of concurrent programs.

– *Tree-shaped data* is easy to handle in the aggregate data approach, but is problematic in the channel approaches.

Chapter 4

Computational Shape

The goal of slivers is to decompose monolithic recursive procedures into networks of mix-and-match parts that preserve the operational character of the original procedure. But what is meant by the "operational character" of a recursive procedure?

Intuitively, operational character describes how the computations specified by a procedure unfold over time. Fine-grained aspects of this unfolding include the relative order of particular operations and the profile of the storage consumed by the computation as a function of time. But unfoldings also display coarser-grained patterns that appear again and again. The iterative nature of computations generated by a tail-recursive procedure are one example of such a pattern. Other patterns include nested loops, mutual recursions, and different classes of tree walks. These patterns are important enough to programmers that they will spend often expend considerable energy to construct a program that generates a desirable computational pattern.

This chapter describes the computational patterns generated by recursive procedures. I call these patterns *computational shapes*. Computational shapes provide a basis for describing some of the important operational characteristics of computations that are preserved by the sliver technique.

4.1 Linear Shapes

4.1.1 Linear Tiles

Figure 4.1 shows the abstract structure of a simple linear recursion. It is a stack of repeated boxes that grow increasingly shorter on the way down. I call this ladder-like structure a *linear trellis*. Although the figure suggests an infinite regress, any terminating computation will populate only a finite prefix of the trellis.

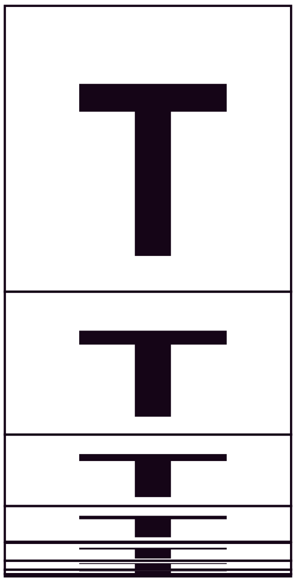


Figure 4.1: Trellis that depicts the structure of a simple linear recursion.

Each **T**-labeled box in the trellis, called a *linear tile*, represents the computation performed by one layer of the recursion. The interface to a tile is shown in Figure 4.2. The top line of a tile, called the *call boundary*, corresponds to the recursive procedure call that initiates the computation in the tile. The bottom line of a tile, called the *subcall boundary*, corresponds to the single recursive *subcall* that a tile computation is allowed to make.[1] Within a linear trellis, the subcall boundary of one tile coincides with the call boundary of the tile below it.

[1]A tile computation might not make any subcall; in fact, a linear recursion will terminate only if some tile in the trellis does not make a subcall. Also, a tile computation is allowed to have several potential subcalls as long as it is guaranteed to actually call at most one of them.

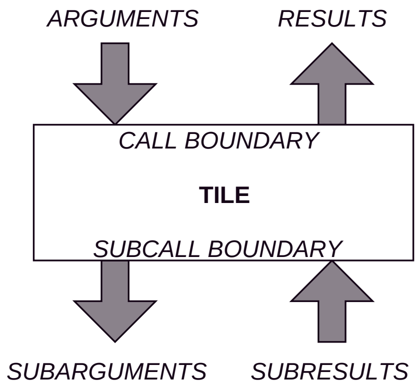


Figure 4.2: Interface to a linear tile. Each shaded arrow designates the transmission of zero or more values.

A tile receives *arguments* and passes *results* across the call boundary. Additionally, it may pass *subarguments* and receive *subresults* across the subcall boundary. Each of these actions (receiving/passing arguments/results) corresponds to an event that the tile computation may participate in:

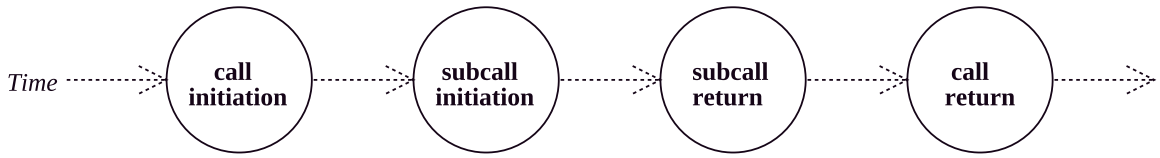
1. *Call initiation* marks the beginning of the tile computation. Since all argument values must be available before call initiation, the call boundary is said to be *strict*.

2. *Subcall initiation* marks the beginning of the subcall's computation. Due to the strictness of the subcall boundary, all subargument values must be computed before subcall initiation.

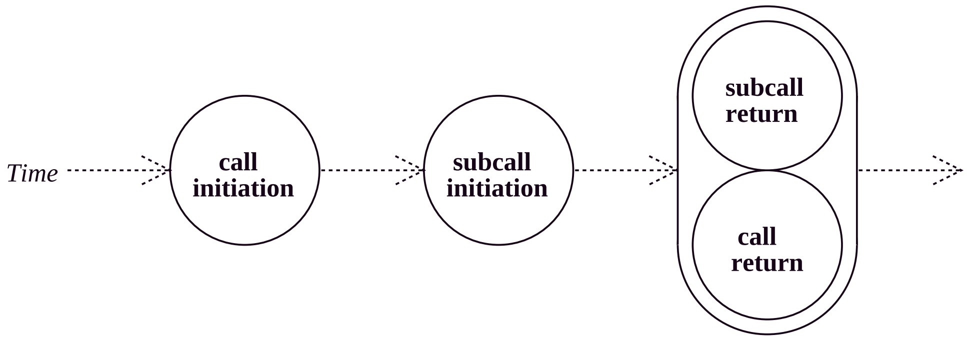
3. *Subcall return* marks the end of the subcall's computation. After this point, the result values of the subcall are available.

4. *Call return* marks the end of the tile computation. The result values returned by the call must be available before this point.

These four events are ordered linearly in time (Figure 4.3). A tile computation that peforms no operations between the subcall return and call return events is said to be *tail-recursive*. In this case, the two events are considered to temporally coincide.



(a) General tile



(b) Tail-recursive tile

Figure 4.3: Time ordering of linear tile events.

4.1.2 Linear Orientation

Operations performed by a tile computation can be classified according to when they can be performed with respect to the subcall:

1. A *down* operation must be performed between call initiation and subcall initiation.

2. An *up* operation must be performed between subcall return and call return.

3. An *across* operation is unordered with respect to subcall initiation and return. It must be performed within the call, but may be performed before, during, or after the subcall.

This classification will be called the *orientation* of an operator. The timing constraints on the three orientations are summarized by Figure 4.4.

4.1.3 Linear Shards

Because the timing constraints are mutually exclusive, they partition a tile's computation into into disjoint collections of operations, which I will call *shards*. The partitioning of a tile

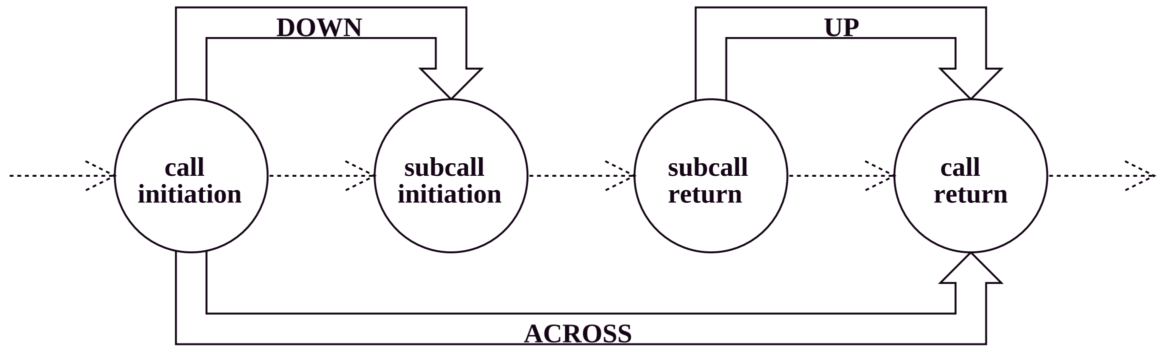


Figure 4.4: Timing constraints that define the orientation of an operation within in a linear tile.

into shards is graphically depicted in Figure 4.5. The *down* shard computes subarguments

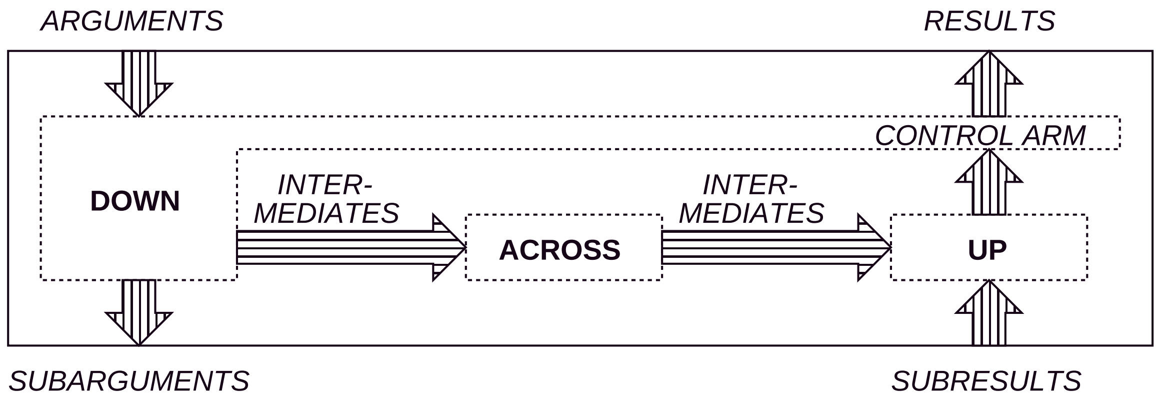
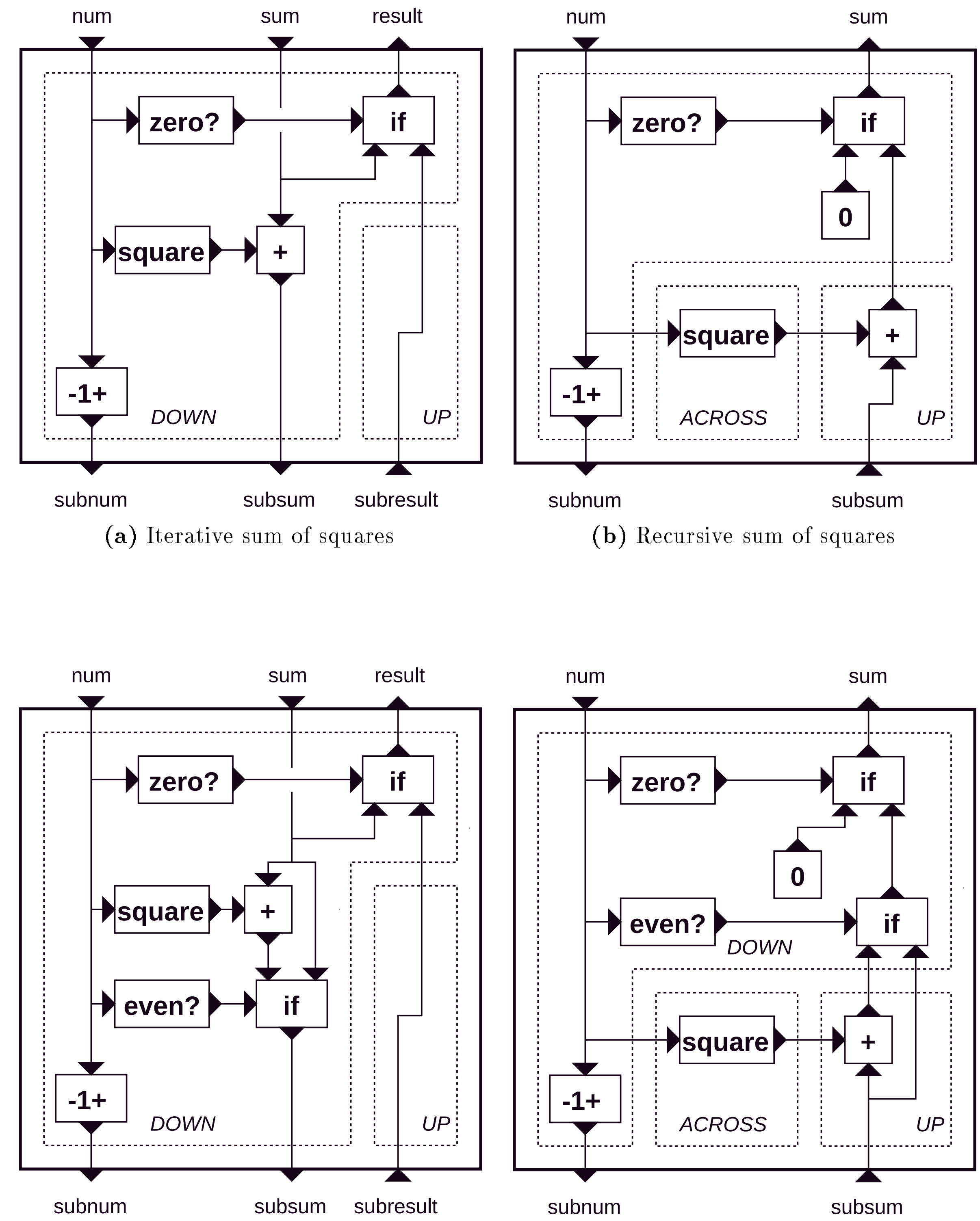


Figure 4.5: Computational shards within a linear tile.

from arguments, while the *up* shard computes results from subresults. Arguments and intermediate values may be transmitted via so-called *intermediates* from the *down* shard, through the *across* shard, and to the *up* shard. In a tail-recursive tile, both the *across* and *up* shards are trivial — they contain no operations. The *down* shard usually has a *control arm* interposed between the *up* shard and the results; this will be explained shortly.

Tiles for some simple computations are shown in Figure 4.6.

The tiles are arranged in pairs of iterative (left) and recursive (right) approaches to the same problem. Tiles **(a)** and **(b)** calculate the sum of the squares of the integers between



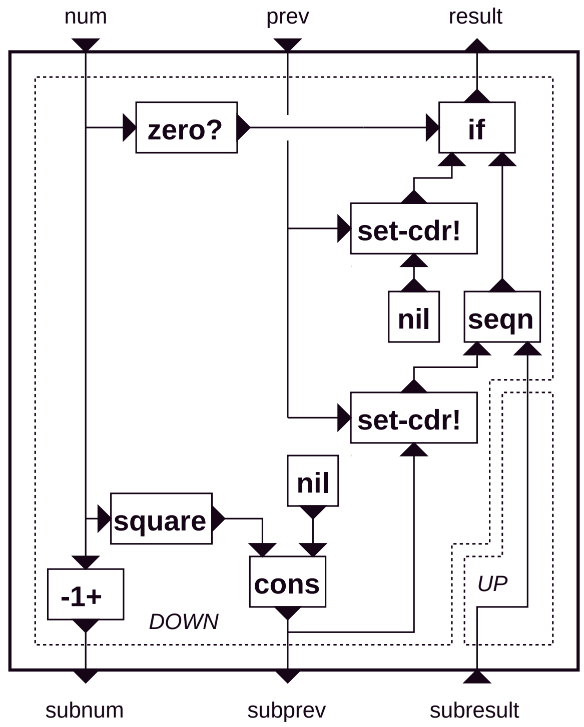
(a) Iterative sum of squares

(b) Recursive sum of squares

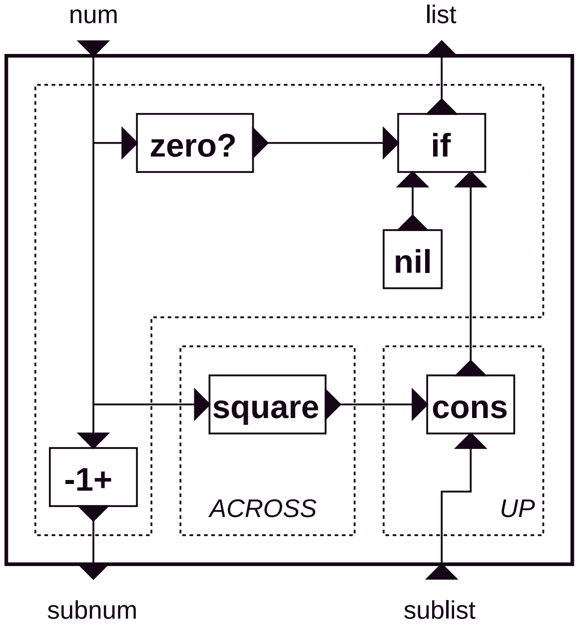
(c) Iterative sum of squared evens

(d) Recursive sum of squared evens

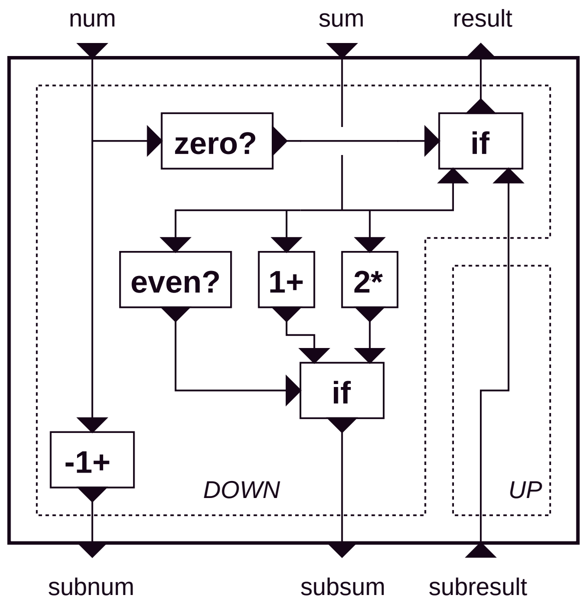
Figure 4.6: (Part I) Sample tiles from some linear recursive computations.



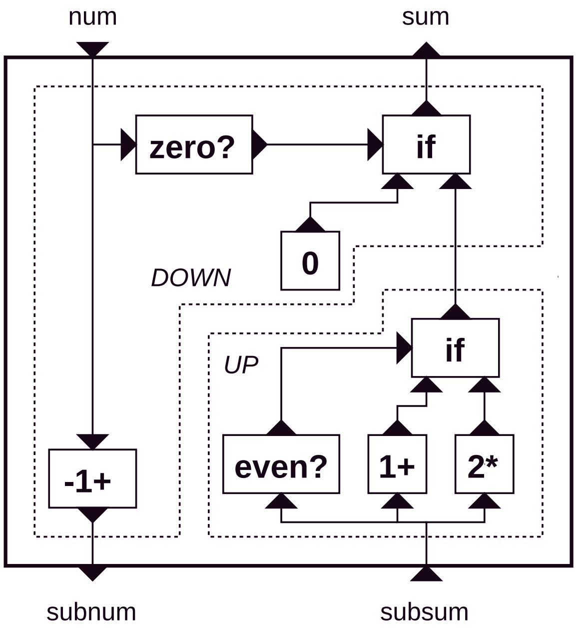
(e) Iterative list of squares



(f) Recursive list of squares



(g) Iterative even/odd



(h) Recursive even/odd

Figure 4.6: (Part II) Sample tiles from some linear recursive computations.

1 and n. Tiles **(c)** and **(d)** calculate the sum of the squares of the even integers between 1 and n. Tiles **(e)** and **(f)** list the squares of the integers between 1 and n. Tiles **(g)** and **(h)** calculate the nth term of the *even-odd* sequence *eo* defined as follows:

$$
\begin{aligned}
eo_0 &= 0 \\
eo_i &= \begin{cases} 1 + eo_{i-1} & \text{if } i > 0,\ eo_{i-1} \text{ even} \\ 2eo_{i-1} & \text{if } i > 0,\ eo_{i-1} \text{ odd} \end{cases}
\end{aligned}
$$

In each case, the tile is the computational unit repeated by the corresponding loop or recursion. For now, issues of initialization (how arguments for the top-level call are specified) and finalization (what is done with the results from the top-level call) will be ignored.

4.1.4 An Operational Model

Building intuitions about tiles and shards requires some understanding of how their computational innards work. The precise details concerning the interpretation of computational elements like those pictured in Figure 4.6 won't be spelled out until the exposition of the EDGAR model in Chapter 8. For now, assume that computation proceeds in a demand-driven fashion as follows:

- Computation within in a tile is initiated by the request of its results. The computation returns only when values are available for all results.

- A requested operator node (e.g. `zero?`, `-1+`, `set-cdr!`) requests all of its subnodes, which are then evaluated concurrently. When values for all subnodes are available, the operator is *performed* (computes a result).

- A requested `if` node first requests the value of its test subnode. When the test value is available, the `if` is *performed* by rerouting the request for its value to either its *then* subnode or its *else* subnode, as appropriate.

- A requested `seqn` node first requests the value of its left subnode. When the left value becomes available the `seqn` is *performed* by rerouting the request for its value to its right subnode. (The value of the left subnode is ignored).

- A request for a subresult produced by a subcall first propagates requests to all of the subargument nodes, which are then evaluated concurrently. Subcall initiation occurs only after *all* subresults have been requested and *all* subargument values are available; i.e., the subcall is strict in all of its arguments. At this point, the subresult requests are allowed to propagate across the subcall boundary to the computation of the tile below.

Based on the above operational model in mind, here are some observations about the sample tiles in Figure 4.6:

- The demand-driven handling of conditionals accounts for the control arm of the *down* shard. Any `if`s that appear in this arm are performed before the subcall, so they belong in the *down* shard. (I assume that an `if` is always performed in the same shard as the operation that determines its test value.) The name "control arm" derives from the fact that these conditionals control the rest of the tile's computation. An `if` can appear outside the control arm when its test is determined by an *up* operation, as in tile **(h)**.

- The operator nodes and subcall boundary represent only potential computation, not actual computation. For example, when the upper-left `if` in any of the sample tiles tests true, the subcall and most of the tile nodes are not performed.

- The *up* shard may be a trivial computation that contains wires but no operations (e.g., tiles **(a)**, **(c)**, **(e)**, and **(g)**). This signifies a tail-recursive tile computation. Intuitively, a tail-recursive tile never returns any results because there is no *up* computation to be performed. Instead, requests for the results of a tail-recursive tile are simply rerouted to become requests for the subresults.

- Some tiles (e.g., tile **(d)**) have both trivial and non-trivial paths through an *up* shard. I call such tiles *conditionally tail-recursive*. Since the tail-recursive property must be known before subcall initiation, any condition on which tail-recursion depends must be tested in the *down* shard.

- The `square` operations in tiles **(b)**, **(d)**, and **(f)** are *across* operations because they must be performed during the tile computation but can be performed before, during, or after the subcall. In contrast, the corresponding `square` operations in tiles **(a)**, **(c)**, and **(e)** are *down* operations because subargument evaluation requires their results. Due to such dependencies, tail-recursive tiles cannot have an *across* shard. *Across* shards that are empty or trivial (i.e., only wires) are simply omitted from tile diagrams.

- In tile **(g)**, the `seqn` node serves to control the order of the side-effects performed by the `set-cdr!` operator. The value returned by the tile doesn't actually matter because the finalizer for the tile (not shown) is responsible for maintaining a pointer to the result.

4.1.5 Linear Tile Shapes

It is useful to characterize tiles according to their shards. In the case of linear tiles, the main distinguishing feature is whether or not the *up* and *across* shards are trivial. A tile with trivial *up* and *across* shards will be said to have a *down shape* because it consists of only a *down* component. A computation with a non-trivial *up* component will be said to have an *up shape*. For now, "*down* shape" and "*up* shape" can be treated as synonyms for "tail-recursive" and "non-tail-recursive". But we will soon encounter other computational patterns that extend the shape notion beyond tail-recursion.

4.1.6 Linear Computations

Individual tiles are stacked together to represent a whole computation. For now, I will only consider computations that can be expressed by replicating the same tile throughout the linear trellis of Figure 4.1. This defines the class of *unitilable* computations.

For example, a recursive sum-of-squares computation on 3 would be constructed out of four instances of tile **(b)** from Figure 4.6 (three for the non-zero cases, and one for the zero case). Operationally, we can imagine that new tiles are dynamically appended to the bottom of a stack only when a subcall is initiated.

A terminating computation uses only a finite number of tiles. Some computations, however, conceptually require an infinite number of tiles. For example, a sum-of-squares computation on a negative input would require unlimited instances of tile **(b)** because the `zero?` test would never be true. Some tiles can *only* generate infinite computations; for example, both tiles in Figure 4.7 give rise to computations that are guaranteed never to return, regardless of their inputs.

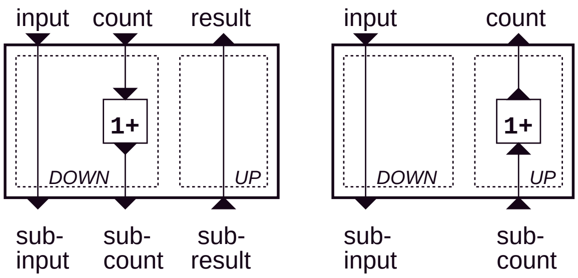


Figure 4.7: Two tiles that generate infinite computations regardless of input.

A particular computation can be classified according to the timing relationships induced among all the call events that occur during the computation. Figure 4.8 shows some timing configurations for linear computations. Each pair of circles represents the initiation (`I`) and return (`R`) events for a single call boundary. A solid directed line from event A to event B indicates that A must happen before B. It is also assumed that every initiation event must occur before its paired return event. A dotted line connecting two events indicates that they occur simultaneously; this is used to represent tail-recursion.

The configurations in the Figure differ in the relationships among the return events. All returns in configuration **(a)** occur at the same time, indicating that all of its calls are tail-recursive. In contrast, in configuration **(c)**, every subcall return occurs before the corresponding call return; this means that no calls are tail-recursive. Configuration **(b)** exhibits both tail-recursive and non-tail-recursive calls.

The timing diagrams suggest that the computation performed within a linear trellis can naturally be divided into *down* and *up* phases. In the *down phase*, the *down* shards of the tiles are performed in a top to bottom order. In the *up phase*, the stack of pending *up* shards are performed in bottom to top order. *Across* operations may happen at any time (subject to data dependencies) between the point where the down phase enters the tile computation

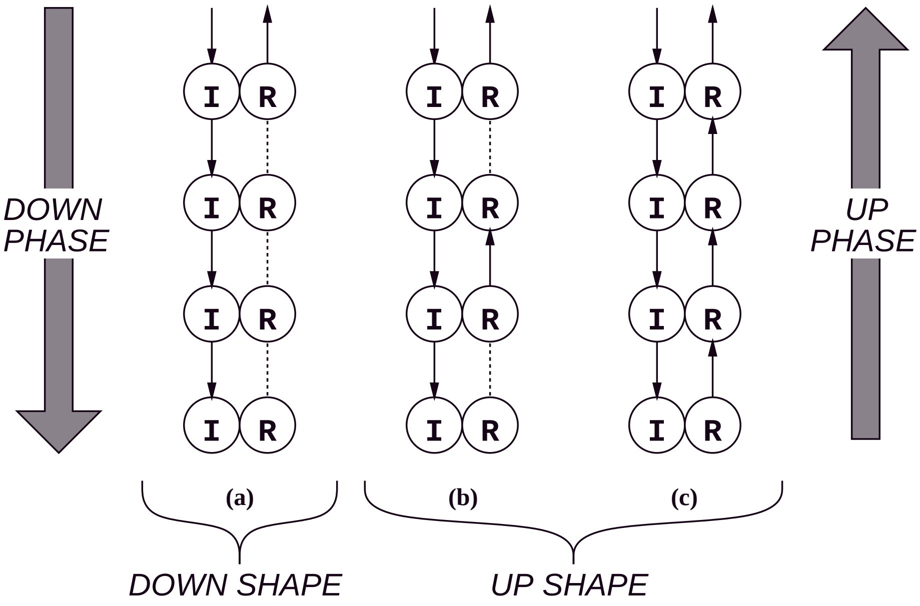


Figure 4.8: Three possible timing configurations for a linear computation.

and the point where the up phase leaves it.

If all the tiles in a linear computation prove to be tail-recursive, the *up* phase will be trivial (i.e., will perform no operations). In this case, the entire linear computation is a pure iteration and is said to have a *linear down shape*. If a linear computation has one or more tiles that turns out not be tail-recursive, then it exhibits some stack-pushing behavior and is said to have a *linear up shape*.

These computational shapes are dynamic properties of running computations. In contrast, the tile shapes defined earlier are static properties of tile structure. The shape of a tile gives a conservative approximation of the shape of a computation generated by that tile. A tile with *down* shape necessarily generates a computation with *down* shape. However, a tile with *up* shape doesn't necessarily generate an *up*-shaped computation. For example, a conditionally tail-recursive tile has *up* shape, but for some inputs may generate iterative computations. In fact, there are *up*-shaped tiles that *always* generate *down*-shaped computations; Figure 4.9 gives one such example. Nevertheless, in practice, tile shape tends to be a good predictor computation shape.

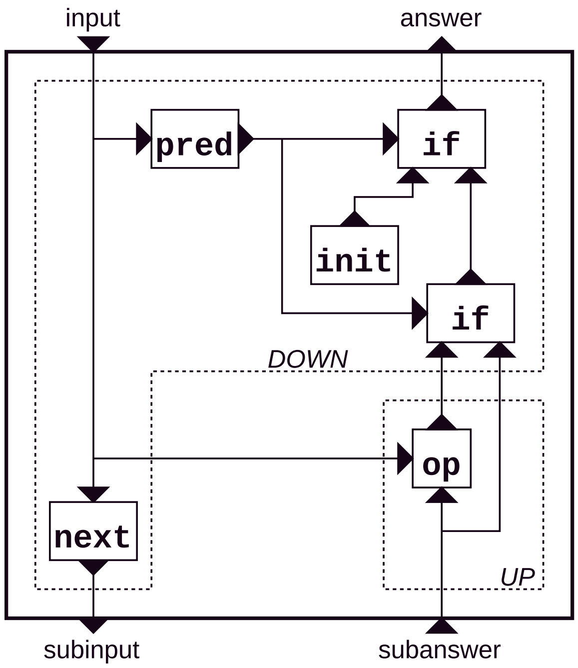


Figure 4.9: A tile with *up* shape that always generates a computation with *down* shape. Because the result of `pred` is tested by both `if` nodes, the *then* branch of the lower `if` node can never be taken.

4.1.7 Wrinkles

Deadlock

There are a few extra issues to discuss before leaving linear shapes. First is the issue of computational fate. So far, we have studied computations that terminate with a result and computations that don't terminate. There is a third possibility: computations that get stuck. Figure 4.10 depicts a tile that generates such a computation. The problem is that `op`

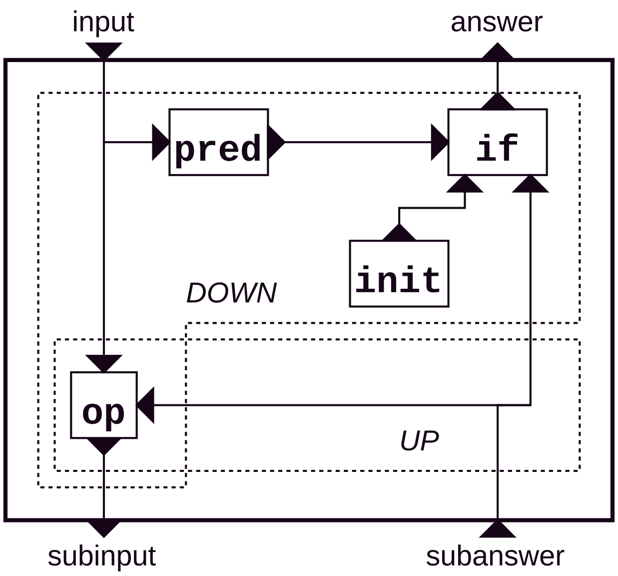


Figure 4.10: A tile that gives rise to a computation that deadlocks.

requires the subresult value in order to compute the subargument value. But the subresult value is not available until the subcall returns, and the subcall cannot be initiated until the subargument is available. This cyclic dependency halts the computation dead in its tracks; the resulting state is called *deadlock*. Deadlock will play an important role in the lock step processing model developed in Chapter 5.

As with computational shape, it is often possible to predict whether or not a computation will deadlock based on the structure of its generating tile. An operation that can be classified in both the *down* and *up* components of a tile (like `op` in the example) is a leading candidate for causing deadlock. But, like most interesting properties, deadlock is uncomputable in general. As an added complication, it turns out that in the presence of non-strict operators, not all cyclic dependencies result in deadlock. I will return to these issues later.

Multiple Potential Subcalls

Second, it is worth mentioning that not all tile diagrams are as tidy as the ones in Figure 4.6. For example, consider a tile (Figure 4.11) that models the following exponentiation procedure:

```
(define (fast-expt_rec base power)
  (if (zero? power)
      1
      (if (even? power)
          (square (fast-expt_rec base (/ power 2)))
          (* base (fast-expt_rec base (- power 1))))))
```

A different `power` argument is computed in each of the branches determined by the `even?` test. So there are two potential subcalls, at most one of which can be taken. This is represented in the tile diagram by two subcall boundaries with an **OR** separator. This is an ad hoc and awkward way of "sharing" several subcalls along the subcall boundary lines. This situation could be improved by decomposing the conditional into *split* and *join* operations (e.g., see [Den75]) that would enable the tile to share a single subcall boundary among several potential calls. However, this "problem" is purely an issue of visual presentation; in terms of the computational model, all that matters is that a tile computation make at most one subcall.

Initialization and Finalization

Third, the notion that many recursive computations can be constructed by replicating a single tile ignores important issues of initialization and finalization. For example, the *down*-shaped tiles **(a)**, **(c)**, and **(e)** from Figure 4.6 nowhere specify that the initial sum accumulation value should be 0. Similarly, the cdr-bashing list accumulation tile (**(g)**) does not spell out the important details of how to start and finish the computation. For these purposes, I will assume the existence of unreplicated *interface tiles* that sit between trellis tiles and the rest of a computation. Figure 4.12 illustrates interface tiles for the summation and cdr-bashing examples.

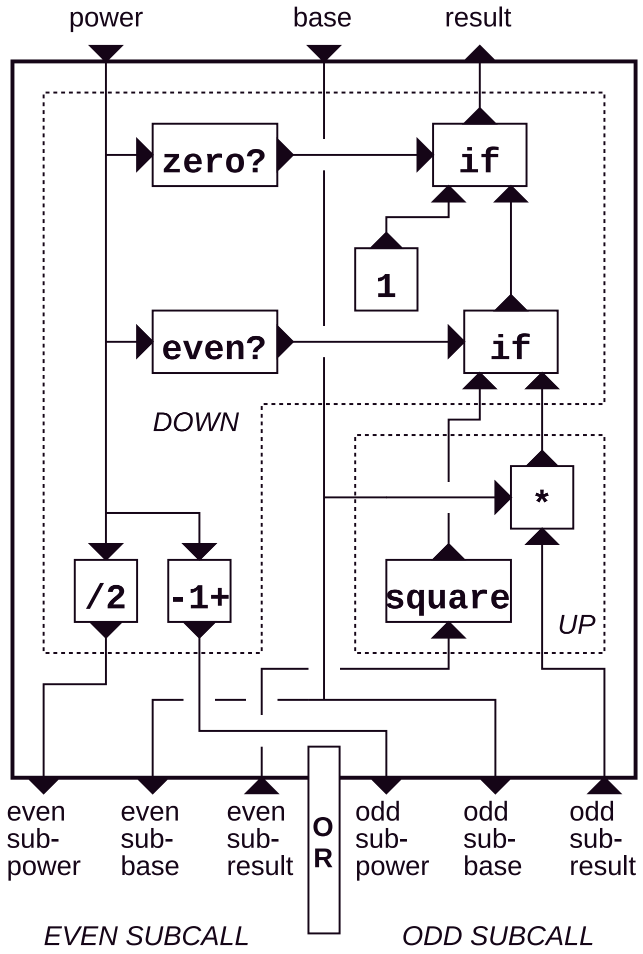
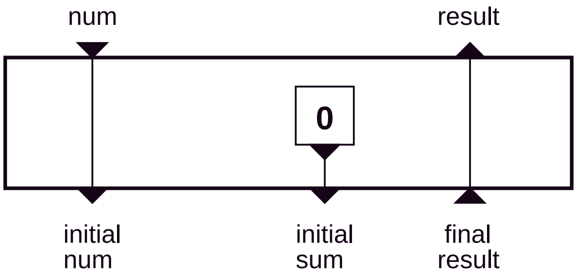
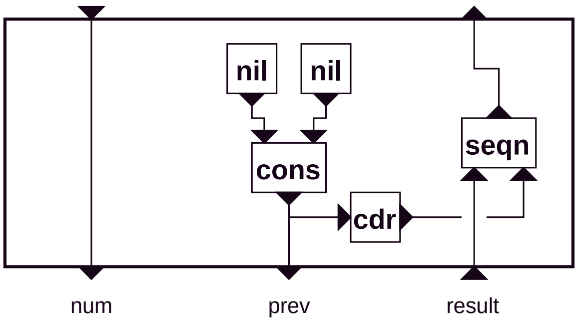


Figure 4.11: A tile with two potential subcalls, at most one of which can be initiated.



(a) Interface tile for iterative summing tiles.



(b) Interface tile for the cdr-bashing tile.

Figure 4.12: Interface tiles to a summation iteration and a cdr-bashing copying routine.

More Complex Recursion Patterns

Finally, I have only described a very narrow range of linear recursions — namely those that can be represented by the replication of a single tile. More complex linear computations, like mutual recursions and nested loops, can have structures that are composed out of several different kinds of tiles (see Figure 4.13). For now, I will continue to ignore these complexities and stick to the simple unitilable case. Nevertheless, the sliver technique based on this theory of shape will be able to handle more complex patterns of recursion.

4.2 Tree Shapes

The shape concepts developed for linear computations generalize to tree-structured computations. For simplicity, I will only consider binary computation trees for now. However, the concepts developed here can be extended to more general computation trees in which the branching factor is a non-uniform function of each tree node.

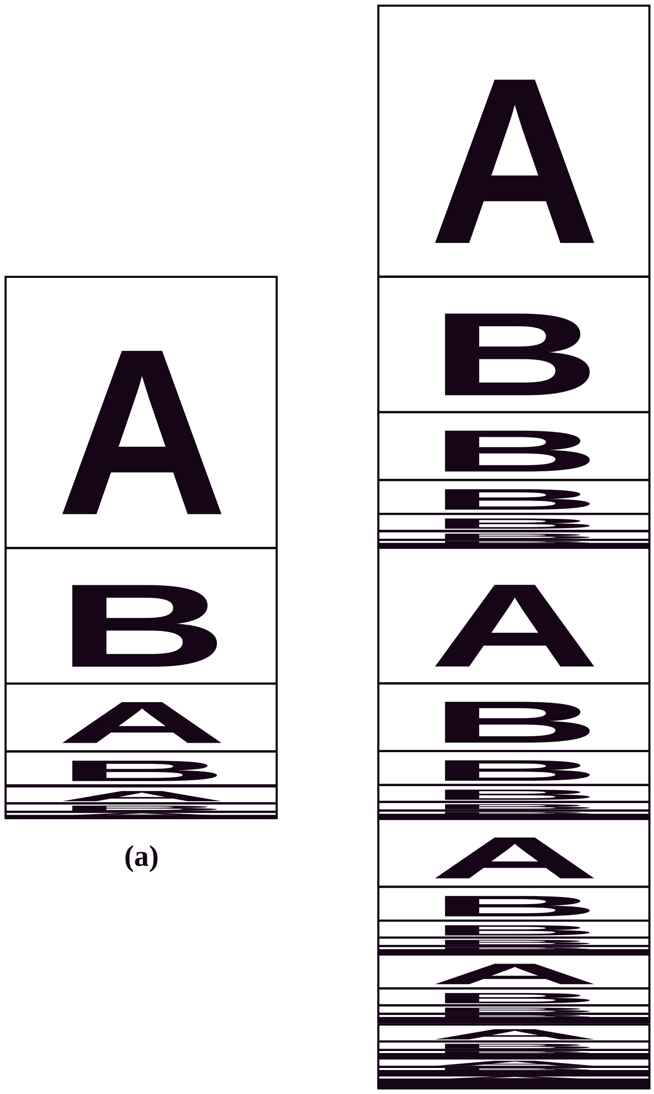
4.2.1 Binary Tiles

Figure 4.14 depicts a *binary trellis* that shows the structure of a simple recursive tree computation populated with instances of the *binary tile* labelled **T**. The interface to a binary tile is shown in Figure 4.15. The chief difference between binary tiles and linear tiles is that a binary tile may make up to two subcalls whereas a linear tile can make at most one. This leads to an interface with *left* and *right* subcall boundaries. The **AND** separator in Figure 4.15 indicates that both subcalls may be initiated. In contrast, the **OR** separator mentioned in the previous section indicates that at most one subcall may be initiated.

Associated with the computation of a binary tile are initiation and return events for the call boundary and each of the two subcall boundaries. These six events exhibit the branching time partial order illustrated in Figure 4.16.

4.2.2 Binary Orientation

Since an operation has a *down*, *across*, or *up* orientation with respect to each of the two subcall boundaries, there are nine distinct orientations for a binary tile operator. These are



(a)

(b)

Figure 4.13: Some linear trellises constructed from two tiles. (a) is intended to suggest mutual recursion between **A** and **B**, while (b) is intended to suggest an inner loop of **B**s nested within an outer loop of **A**s.

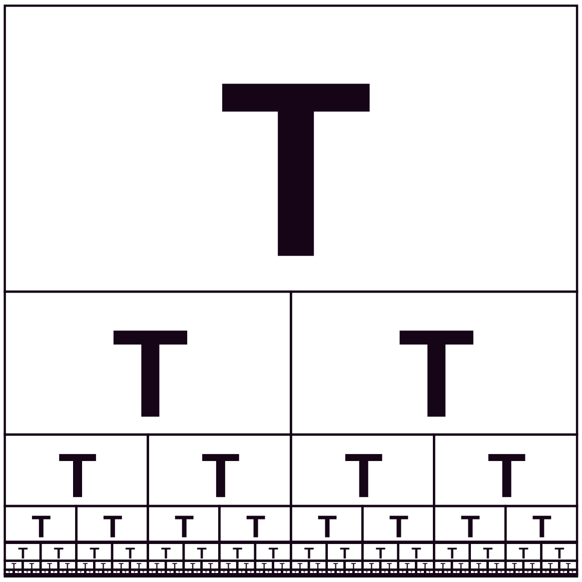


Figure 4.14: Trellis that depicts the structure of a simple tree recursion.

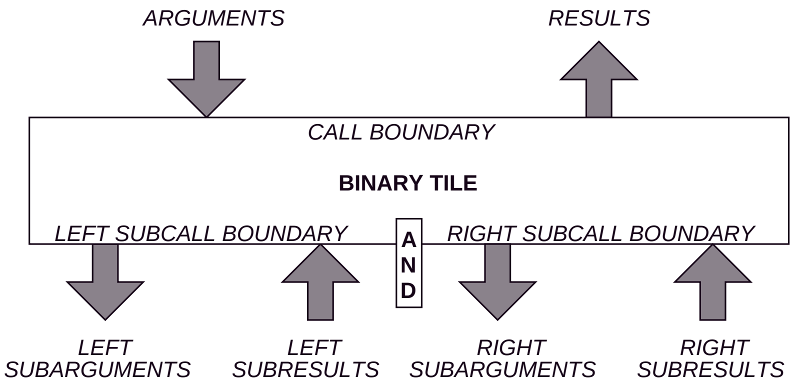


Figure 4.15: The interface to a binary tile.

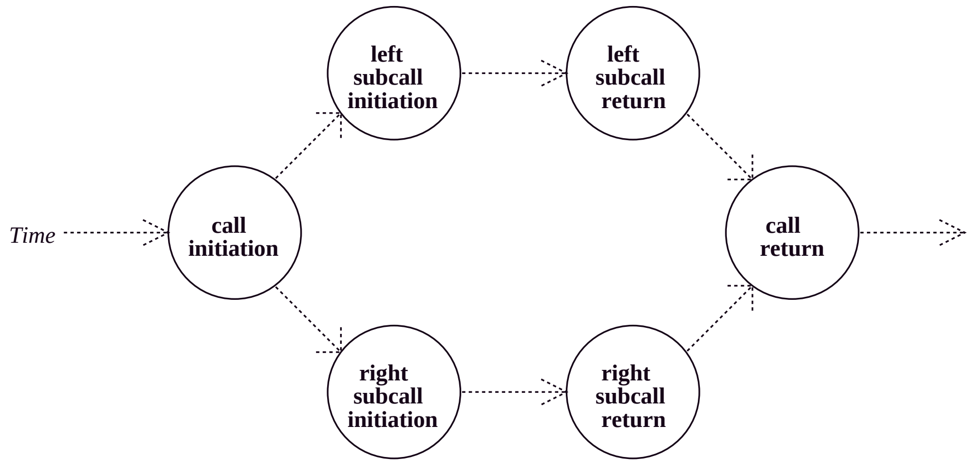


Figure 4.16: Time ordering of binary tile events.

enumerated in table 4.1. Distinguishing nine orientations may seem like overkill, but we will see that it provides us with a terse vocabulary for describing some important operational traits.

Figure 4.17 graphically summarizes the timing constraints for the nine binary orientations. The *down-both* orientation involves three events because an operation with this orientation must occur after call initiation but before both subcall initiations. The *up-both* orientation is similar in this regard. In contrast, the *down-left* orientation involves only two events. This indicates that a *down-left* operation not only must occur before initiation of the left subcall, but also must be unordered with respect to the right subcall. Otherwise it would be classified as *down-both* or *between-RL*. Similar remarks hold for the *down-right*, *up-left*, and *up-right* orientations. The binary *across* orientation has the same meaning as the linear one; an *across* operation can be performed at any point within the duration of the tile computation.

Sometimes it is convenient to use a less detailed vocabulary for talking about binary orientations. Here are a few helpful abstractions:

- *between-LR* and *between-RL* will be classified as *between* orientations.
- *down-both*, *down-left*, and *down-right* will be classified as *down* orientations.

Left Subcall Orientation	Right Subcall Orientation	Binary Orientation
down	*down*	*down-both*
down	*across*	*down-left*
down	*up*	*between-RL*
across	*down*	*down-right*
across	*across*	*across*
across	*up*	*up-left*
up	*down*	*between-LR*
up	*across*	*up-right*
up	*up*	*up-both*

Table 4.1: The nine orientations of an operation within a binary tile computation.

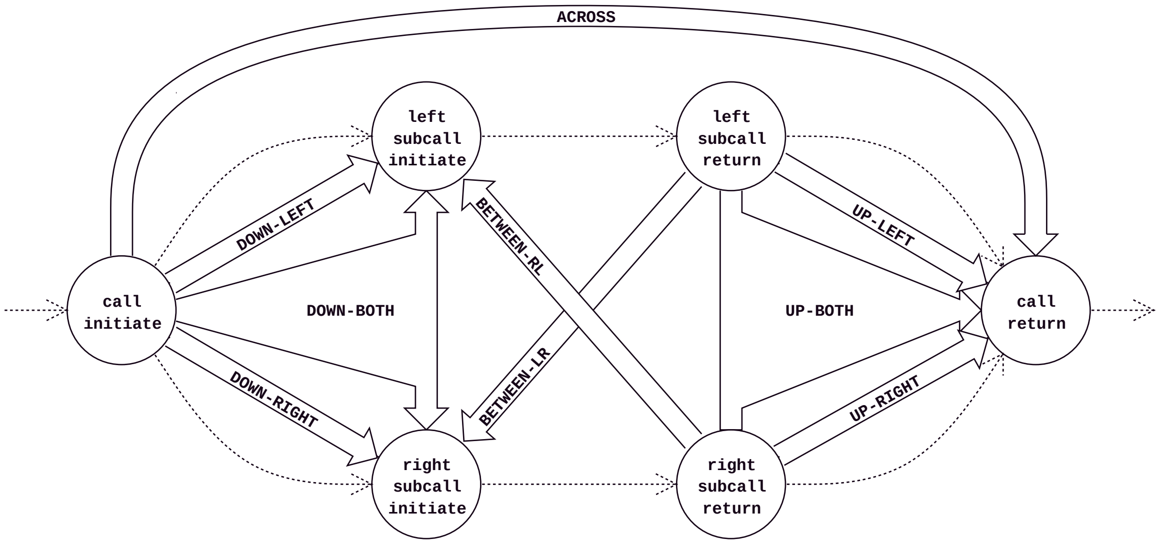


Figure 4.17: Timing constraints that define the orientation of an operation within in a binary tile. The two-headed *down-both* and two-tailed *up-both* arrows indicate constraints that involve three events.

- *up-both*, *up-left*, and *up-right* will be classified as *up* orientations.

Down and *up* for tree orientations can be viewed as generalizing their meaning for linear orientations. Context will distinguish whether linear or tree orientations are intended.

4.2.3 Binary Shards

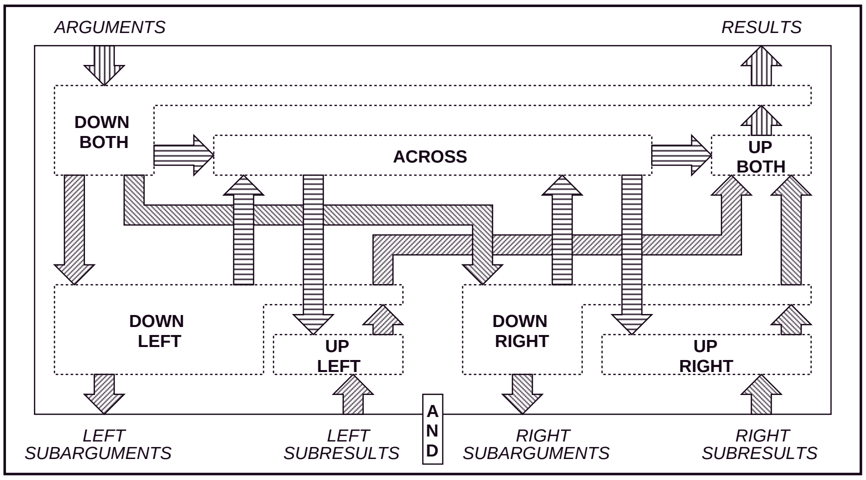
As in a linear tile, the operations within a binary tile can be partitioned into shards according to their orientation. Different possibilities for dataflow between the two subcall boundaries lead to three patterns for binary tile computations (Figure 4.18):

1. A *parallel* tile allows no dataflow between subcalls. Due to the concurrency inherent in the computational model sketched earlier, the subcalls can be evaluated in parallel.
2. A *left-to-right* tile allows the right subarguments to depend on the left subresults. Due to the strictness of a subcall boundary, this means that the left subcall must return before the right subcall is initiated.
3. A *right-to-left* tile is symmetric with the left-to-right one.

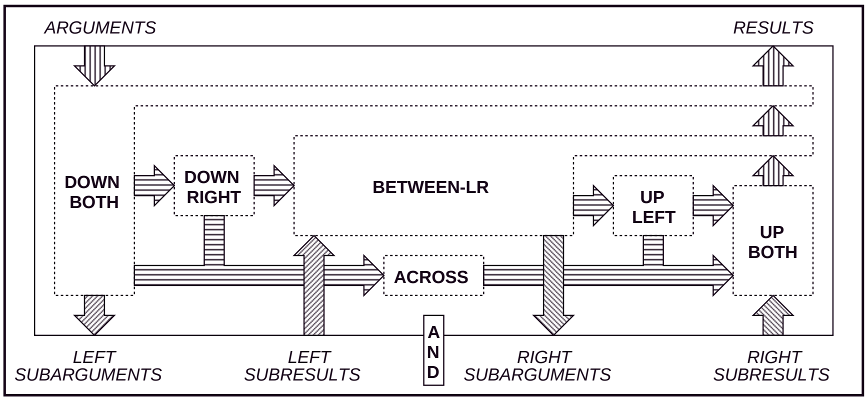
The fourth possibility, a *mutually dependent* tile in which the left and right subcalls depend on each other, is disallowed because the cyclic dependencies would lead to deadlock.[2] The left-to-right and right-to-left tiles are classified as *sequential* tiles because they force one subcall to return before the other is initiated.

Note that each of the three patterns in Figure 4.18 is necessarily missing several shards. The *between-LR* and *between-RL* shards are mutually exclusive because they imply that the subcalls happen in different orders. Clearly, no tile can contain both of these shards; and a parallel tile, whose subcalls must be unordered, can contain neither of them. A left-to-right tile invariably forces every operation that happens before the left subcall to also happen before the right subcall as well. In this case, a *down-left* shard is impossible because its

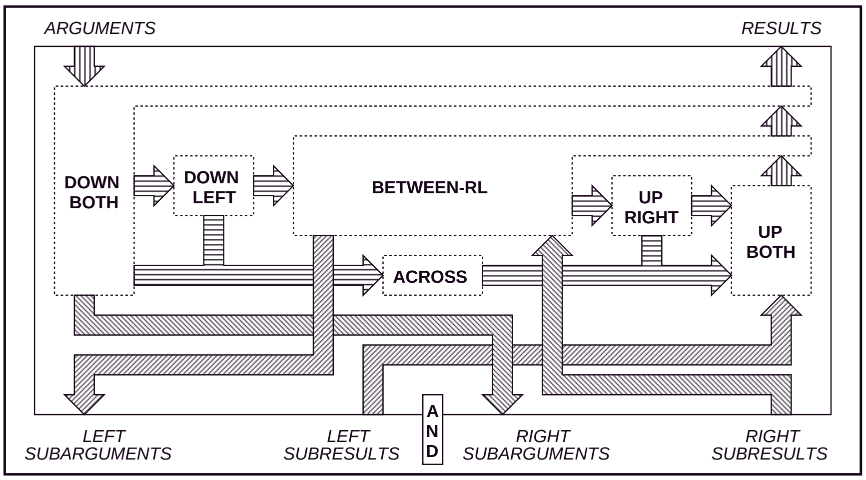
[2] I am assuming here that all operations performed in a tile computation are themselves strict. In the presence of lazy operators, cyclic dependencies are not only possible but often desirable (e.g., see [Bir84], [Joh87]). Later, I will introduce a form of laziness that, if used indiscriminantly, would invalidate the claims made about the impossibility of mutually dependent tiles and the execution order of sequential tiles. However, I will carefully restrict laziness in order to preserve these claims.



(a) Parallel Tile



(b) Left-to-right Tile



(c) Right-to-left Tile

Figure 4.18: Three general shard patterns for a binary tile computation.

operations would have to be unordered with respect to the right subcall; an *up-right* shard can similarly be discounted. Symmetric remarks hold for the right-to-left tile.

Some sample binary tiles appear in Figure 4.19. All of them share a generating fragment (consisting of the doubler `2*` and the incrementer `1+`) that creates a "virtual" binary tree of a given size in which each non-leaf node is numbered with its position in a left-to-right breadth-first traversal. I will call this a *breadth index tree* (see Figure 4.20).

The breadth index tree is virtual in the sense that it never exists as a bona fide data structure; rather, its elements are created and used by the tree-structured computation. Tile **(a)** is a parallel accumulator that collects the virtual tree elements into a tree-shaped data structure in which every node is represented as a list of its element and its left and right subtrees; `nil` represents the empty subtree. Tile **(b)** (**(c)**, **(d)**) collects a list of elements visited during a left-to-right pre-order (in-order, post-order) walk of the virtual tree.[3]

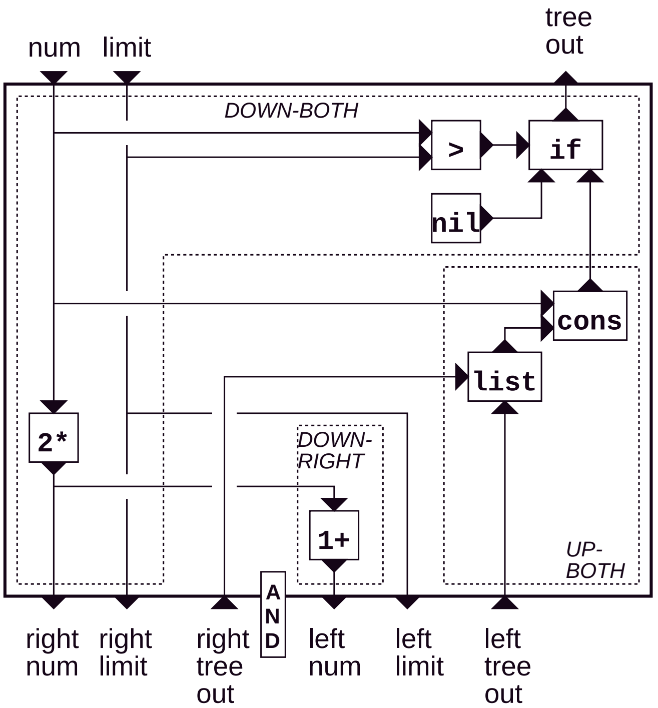
Several shards in the tile patterns of Figure 4.18 have control arms. In general, a control arm can be exhibited by any shard that must occur before some subcall — i.e., *down-both*, *down-left*, *down-right*, *between-LR*, and *between-RL* shards. For example, Figure 4.21 shows a tile with a control arm on both the *down-both* and *between-LR* shards. These allow the accumulation to terminate on entry to the call as well as between the two subcalls. A subtle constraint is that while *down-left* and *down-right* shards may have control arms in a parallel tile, they may not have control arms in a sequential tile.

4.2.4 Binary Tile Shapes

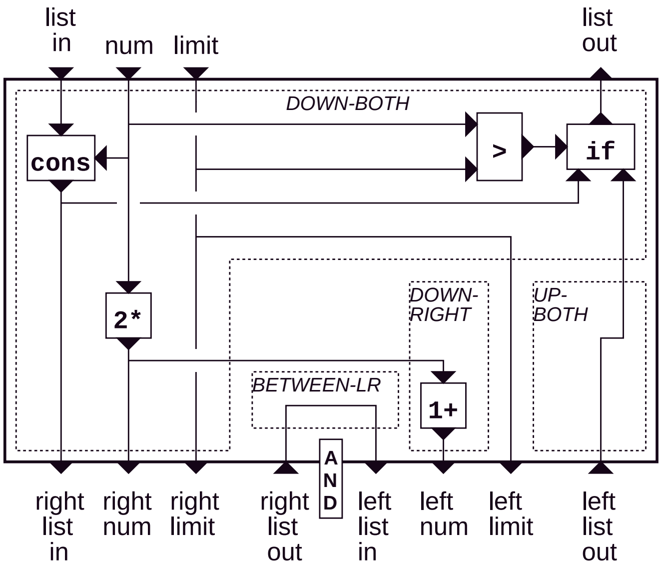
It is helpful to give names to special cases of the binary tile patterns introduced above. As in the linear case, these will be called shapes. A listing of important binary shapes appears in table 4.2.

Binary up tiles are a generalization of *linear up* tiles. These are tiles that simply combine the results of two independent subcalls; Figure 4.19**(a)** is an example. *Binary down* tiles

[3] Note that because `cons` builds the list from the end, the resulting list contains the elements in *reverse* left-to-right pre-order. A left-to-right pre-order list of tree elements can be generated by `cons`-accumulating elements in a right-to-left post-order walk of the tree. Alternately, the cdr-bashing strategy can be used to collect the list in pre-order during a pre-order walk. Similar remarks hold for the in-order and post-order cases.

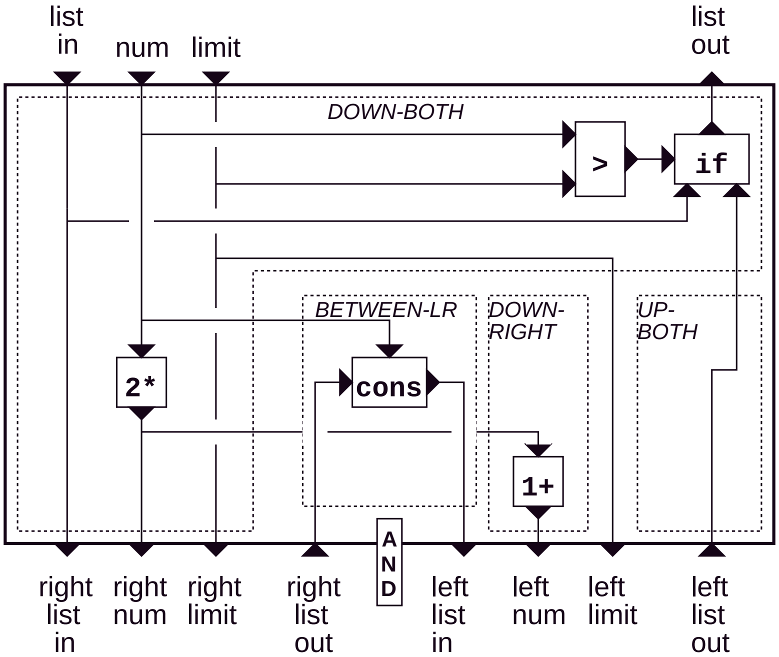


(a) A tile collecting the elements of a virtual tree into a tree-shaped data structure.

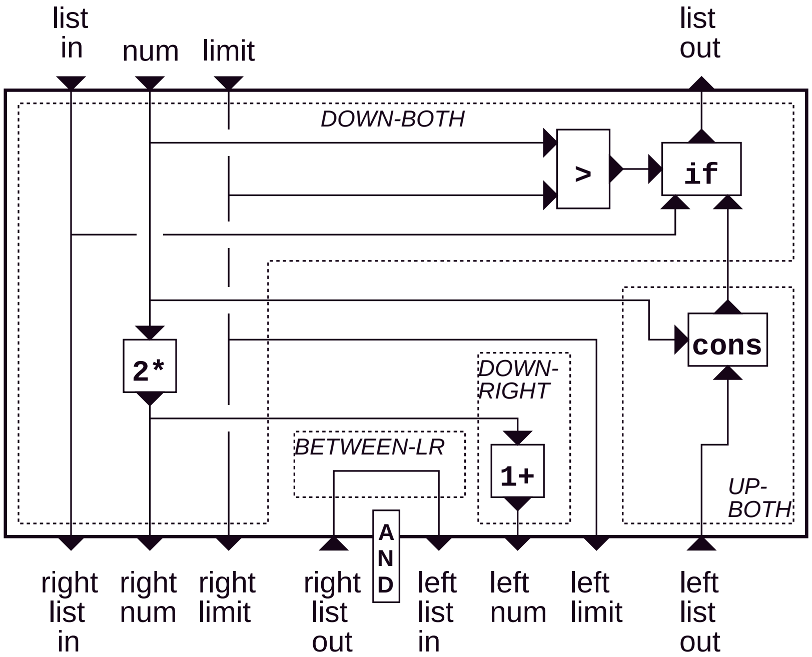


(b) A tile collecting a list of elements during a pre-order walk over a virtual tree.

Figure 4.19: (Part I) Sample binary tiles.



(c) A tile collecting a list of elements during a in-order walk over a virtual tree.



(d) A tile collecting a list of elements during a post-order walk over a virtual tree.

Figure 4.19: (Part II) Sample binary tiles.

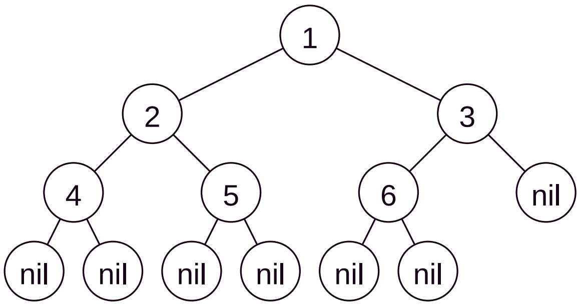


Figure 4.20: A breadth index tree with six elements.

Shape	Binary Pattern	Restrictions
binary down	parallel	*up-both*, *up-left* and *up-right* trivial
binary up	parallel	*up-both* non-trivial
LR pre	left-to-right	*between-LR* and *up-both* trivial
LR in	left-to-right	*between-LR* non-trivial and *up-both* trivial
LR post	left-to-right	*up-both* non-trivial
RL pre	right-to-left	*between-RL* and *up-right* trivial
RL in	right-to-left	*between-RL* non-trivial and *up-both* trivial
LR post	right-to-left	*up-both* non-trivial

Table 4.2: Shapes of binary tiles.

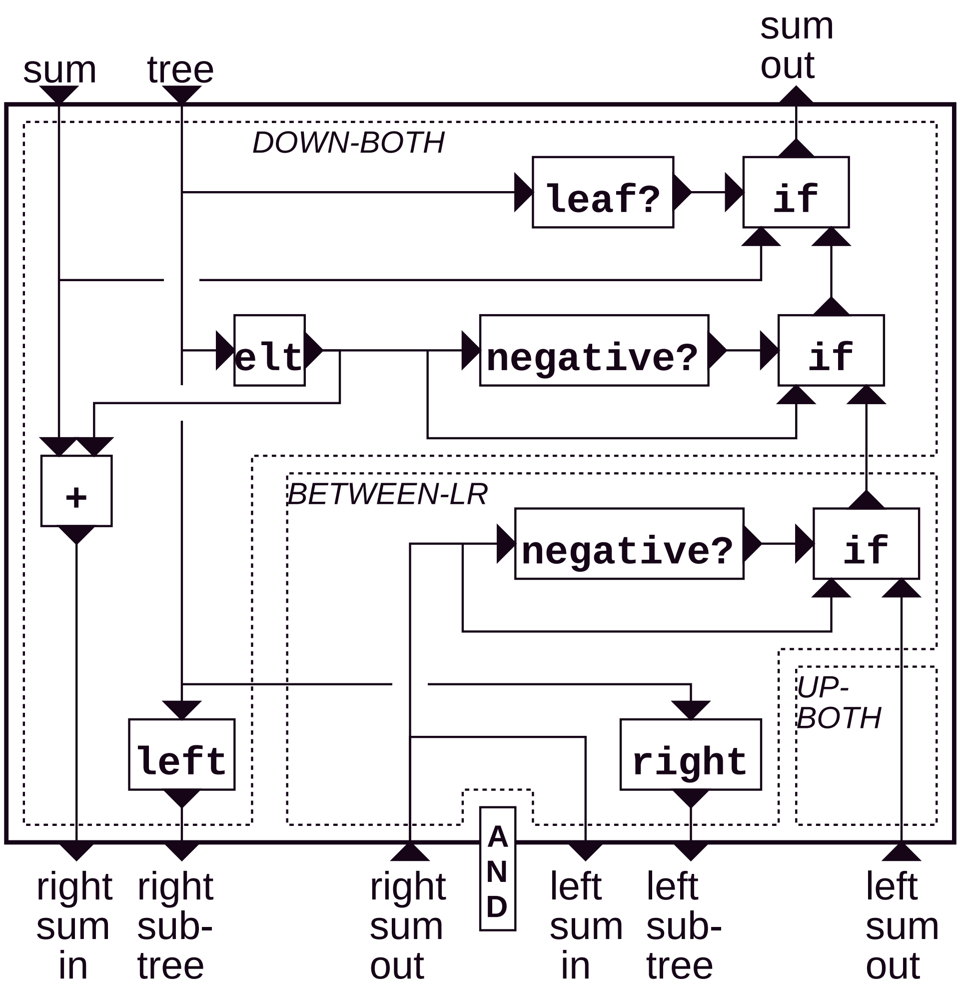


Figure 4.21: A tile in which both the *down-both* and *between-LR* shards have control arms. The `leaf?` operator tests for an elementless leaf node; the `elt`, `left`, and `right` operators return, respectively, the element, left subtree, and right subtree of a non-leaf node. If all the tree elements are non-negative, the tile generates a computation that returns their sum. However, if some of the elements are negative, the tile generates a computation that returns the first negative element encountered in a left-to-right pre-order traversal of the tree.

are more difficult to exhibit, and I postpone them for the moment.

In a *pre* tile, all operations are performed before the first subcall. An *in* tile allows operations between the two calls, while a *post* tile permits operations after the second call. Tiles **(b)**, **(c)**, and **(d)** in Figure 4.19 are examples, respectively, of *pre*, *in*, and *post* tiles. The shape names were chosen to reflect the kind of tree walk implied by the computations. Since the walks can be performed in either a left-to-right or right-to-left direction, the shape names are parameterized by this direction.

The sequential shapes can be ordered by specifity as follows:

$$pre < in < post$$

The shapes are defined so that every sequential tile has a unique most specific shape. These extend the simple *down* $<$ *up* ordering for linear tiles. In fact, note that both *pre* and *in* tiles are effectively tail-recursive in the second subcall.

Binary tile shapes essentially specify the dependencies between different parts of a tree computation. In this respect, they resemble *attribute grammars*, a declarative formalism in which programs can be specified in terms of the information dependencies between the nodes of a grammar-induced tree [DJL88]. (See Section 3.3.3 for a discussion of attribute grammars.)

4.2.5 Binary Down Tiles and Non-strictness

We now return to the notion of *binary down* tiles. These are tiles whose *up-both*, *up-left* and *up-right* shards are all trivial. But using the simple operations introduced so far, it is impossible to construct a binary tile with these properties! Intuitively, the only way to propagate requests to both of the subcalls is through a binary operator. But all the binary operators seen so far would have to be placed in the *up-both* shard because they cannot be peformed until their input values are available.

This problem can be circumvented by introducing a special binary node, `fork2`. A `fork2` node responds to a request by propagating requests to its two subnodes and then immediately returning the boolean true value (`#t`) without waiting for a result from either subnode. The tile in Figure 4.22 shows a simple use of `fork2` to print the elements of a tree. The `seqn` forces a node's element to be printed before any elements in either of its

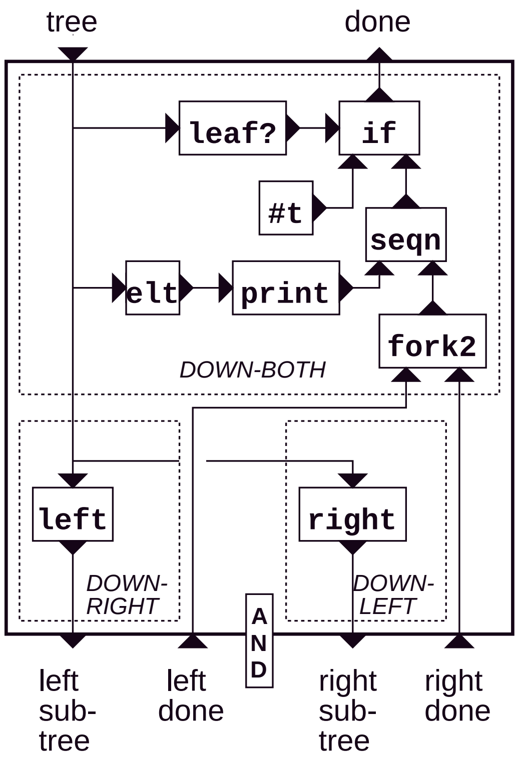


Figure 4.22: A tile with *binary down* shape that prints the elements of a tree. The `print` operator prints its input and then returns `#t`. `Seqn` forces a node's element to be printed before any element in its subtrees. `Fork2` returns after initiating both subcalls without waiting for their results.

subtrees. The `fork2` initiates the printing (in parallel) of the left and right subtrees.

`Fork2`'s strategy of evaluating arguments in parallel with returning a result is called *eager evaluation*. Eager evaluation is an example of a *non-strict* evaluation strategy, so called because the operator is performed before its arguments are completely evaluated. Another form of non-strictness is *lazy evaluation*, in which argument evaluation is suspended until it is required. The lazy data technique introduced in Section 3.1.5 is an example of the lazy evaluation strategy.

The non-strict behavior of `fork2` can change the timing constraints among the call and subcalls. A call to the tree-printing tile may return at any time with respect to the initiation or return of the subcalls (see Figure 4.23). This implies that tree elements may continue to print after the top-level call to the tree-printing tile has returned!

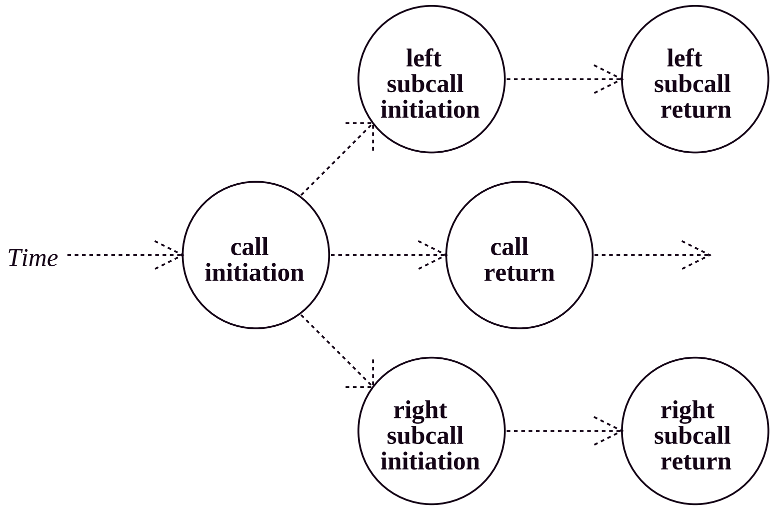


Figure 4.23: The use of the non-strict `fork2` operator changes the timing diagram for the events in a binary tile.

The ability of a non-strict computation to return before completing its computation is a powerful feature for expressing parallelism and speculative computation (see [Tra88, Hal85, Mil87]). Indeed, we will see in Chapter 7 that non-strictness is an essential technique for modularizing computations. However, because it can change timing relationships among calls and other operations, non-strictness complicates reasoning about programs (e.g., see [HA87]). For example, consider a sequential tile in which a non-strict operator links the result of one subcall to the argument of the other subcall. Then it is no longer true that the first subcall must return before the initiation of the second subcall! Due to these kinds of complications, I restrict the use of non-strict operators. Except for `if` and `seqn` (which are non-problematic instances of non-strict operators), I will assume that tile computations do not contain any non-strict operators.

4.2.6 Binary Computations

Under the unitilable assumption, a binary computation can be generated by dynamically replicating a binary tile throughout a binary trellis. As in the linear case, the resulting computations can be classified by computational shape — i.e., according to the timing relationships induced on their call events.

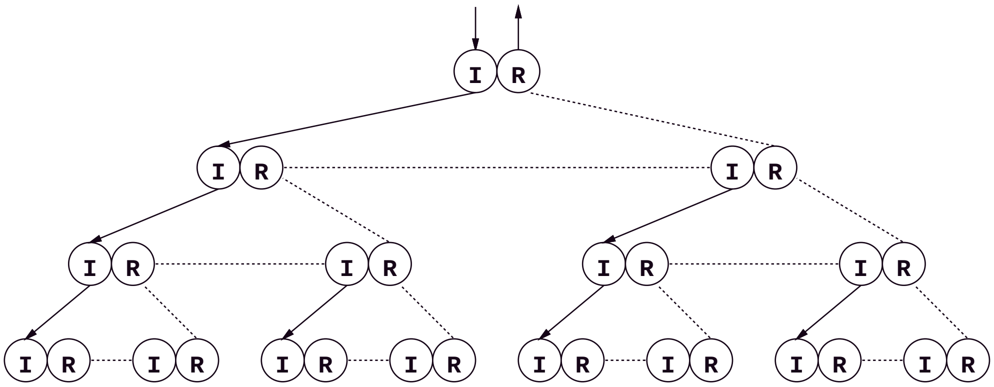
Figure 4.24 shows the three canonical shape configurations for a computation generated by a left-to-right sequential tile. These differ according to which events must happen simultaneously. Strings of events connected by dotted lines indicate paths along which no computation is allowed; all the events on the path occur at the same time. In a *pre*-shaped computation, every return must be connected by such a path to either an initiation event for a right subcall or the return even of the top-level call. A computation with *in* shape additionally forces at least one left subcall return to precede a right subcall initiation, while a *post*-shaped computation is characterized by a subcall return that precedes a call return. Due to the conditional nature of tiles, a *post* tile might actually generate any of the three computational shapes. However, a *pre* tile necessarily generates only *pre* computations, while an *in* tile can may generate only *in* and *pre* computations.

Two possible parallel shapes are illustrated in Figure 4.25. These are distinguished by the relationships among the return events. In a *binary down* computation, all return events are totally unconstrained with respect to each other, while in a *binary up* computation, some subcall returns are required to precede the return of the corresponding call. Note that the assumption that tiles do not contain non-strict operators effectively eliminates the *binary down* shape for tiles. However, by judicious use of non-strictness, it is possible to design computations having this shape.

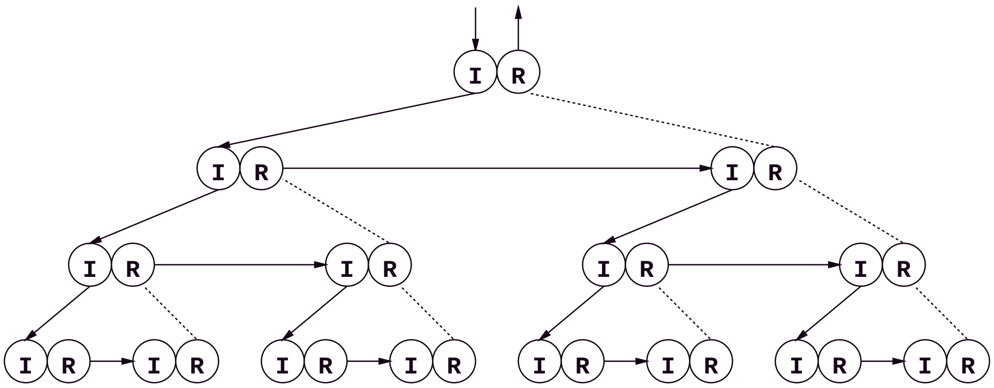
Unlike linear computations, binary computations do not decompose nicely into single *down* and *up* phases. In the sequential case, after a computation walks down one branch of the event tree, it must generally walk back up the branch in order to get to the next branch. In the parallel case, the concurrent evaluation of subcalls means that *down-both* operations in one branch may be interleaved with *up-both* operations in another branch.

The timing diagrams in Figures 4.24 and 4.25 underscore the difference between sequential and parallel computations. At the coarse-grained resolution of the timing diagrams, sequential shapes constrain time to follow a single path through the computation.[4] Because

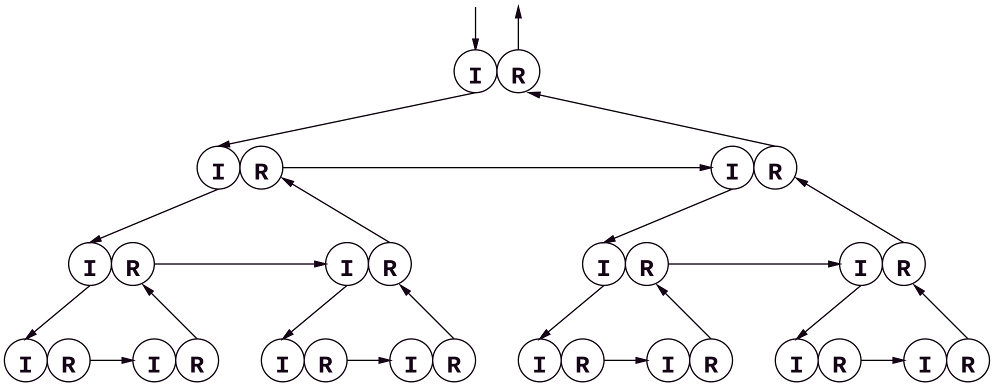
[4]A finer-grained analysis of individual operations might expose local branching of time. For example, an *across* operation in a call can be interleaved with any operations in both subcalls. The timing diagrams supress this level of detail.



(a) Computation having *pre* shape.

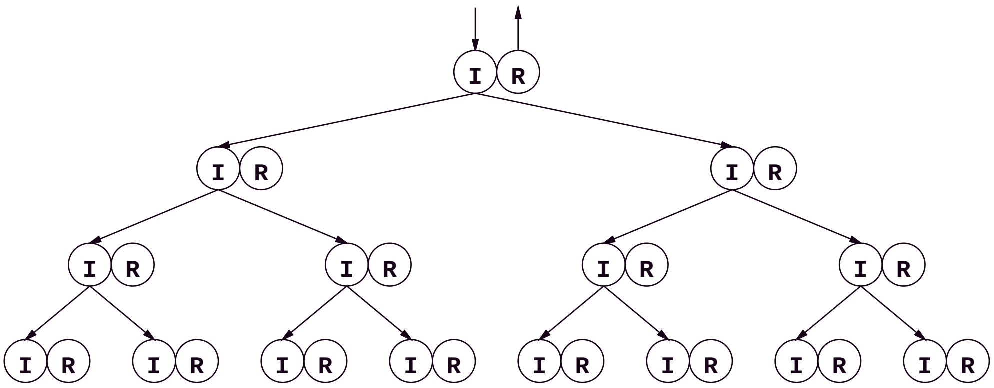


(b) Computation having *in* shape.

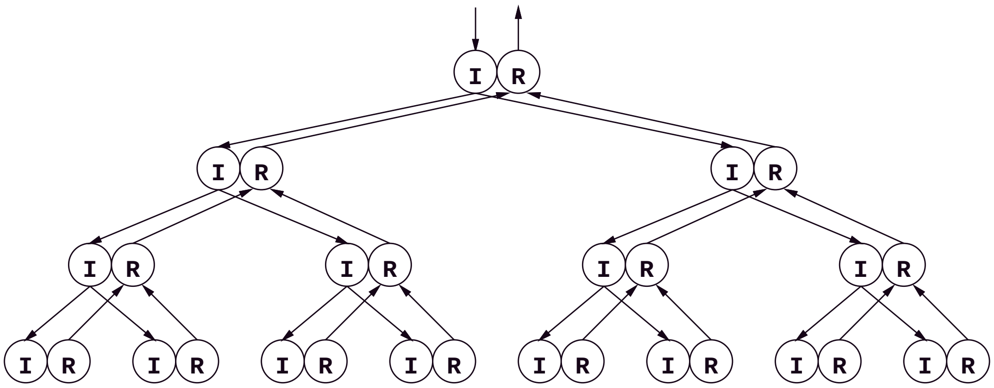


(c) Computation having *post* shape.

Figure 4.24: The timing configurations for sequential computations exhibit a single-threaded structure.



(a) Computation having *binary down* shape.



(b) Computation having *binary up* shape.

Figure 4.25: The timing configurations for parallel computations exhibit a multi-threaded structure.

such a computation effectively has a single locus of control, it is said to be *single-threaded*. Since the branching time nature of parallel computations permits multiple loci of control, they are said to be *multi-threaded*.

While the potential for evaluating subcalls in parallel makes multi-threaded computations attractive candidates for execution on parallel hardware, I will not explore this avenue. Instead, I will focus on fundamental behavioral issues that arise from the branching time nature of multi-threading. In particular, I will be concerned with the following two issues:

1. *Non-Determinism*: In the presence of side-effects, multi-threaded computations may be non-deterministic. Different interleavings of mutation operators can give rise to different results. Various atomicity and synchronization techniques (e.g., [Bir89b, Dij68, Bar92]) may be required to appropriately constrain behavior.

2. *Storage Requirements*: Multi-threaded computations have the potential for consuming much more storage than single-threaded ones. Since a single-threaded computation populates at most one branch of a binary trellis at any time, the space consumed by a single-threaded computation is linear in the depth of the trellis. In contrast, a worst-case multi-threaded computation can populate the entire trellis at once — this leads to space exponential in the depth of the trellis. This problem can be addressed by dynamic strategies (e.g., [BL93]) or by giving the programmer fine-grained control over parallelism (e.g., [GM84]).

Although they permit parallelism, multi-threaded computations do not require it. It is always possible to sequentialize a multi-threaded computation by adding additional constraints that one subcall return before another initiates. For example, many languages require procedural arguments to be evaluated in a particular order even when they have no side-effects. I will eschew these spurious constraints and only pay attention to constraints explicity specified by the programmer.

4.2.7 Discussion

Although we have only studied binary tree computations, the notions developed here generalize to tree computations where the branching factor can be node-dependent. In the more general setting, *binary down* and *binary up* shapes become *parallel down* and *parallel up* shapes. Sequential shapes need to be extended to handle the fact that any permutation of the subcalls defines a possible sequential ordering. Except for the new orderings, *pre* and *post* shapes remain essentially unchanged. However, *in* shapes must be extended to express the fact that an *in* operation may happen between any pair of subcalls consistent with a given sequential walk.

Dropping the unitilable assumption would also greatly increase the number of computations that can be expressed. Then it would be possible for a single *in* tile, say, to have *pre*, *post*, and *binary up* tiles for its subcalls. However, for reasons of simplicity, I will stick to the unitilable assumption for tree computations throughout the remainder of this report.

Chapter 5

Synchronized Lazy Aggregates

In Chapter 3, we saw how existing techniques for programming in the signal processing style exhibit tensions between modularity and control. This chapter introduces a new technique, *synchronized lazy aggregates*, that relaxes some of these tensions. This technique makes concrete the notion, first presented in Chapter 2, that slivers are slices through a monolithic recursive computation.

5.1 A Lock Step Processing Model

Synchronized lazy aggregates are based on a model of processing in which slivers compute in a lock step manner to simulate the behavior of monolithic recursive procedure. Here I present a brief motivation for and overview of the lock step model. The rest of the chapter fleshes out the details of how lock step processing can be achieved.

The lock step model is designed to satisfy two important goals:

1. *Operational faithfulness*: A sliver network as a whole should simulate the operation scheduling and space behavior of a monolithic computation. A mechanism may use 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.

We begin by considering the following question: *Why is it so difficult to model the storage behavior and operation scheduling of a monolithic procedure using exisiting SPS techniques?* To answer this question, we first need to see how operational control is achieved in monolithic procedures. Then we can investigate what goes wrong in the modular case. Finally, based on our analysis, we can propose a fix.

5.1.1 Strict Calls Provide Control

The desirable operational properties of monolithic recursive procedures are due to strict procedure calls. Strictness means that all argument values must be computed before the procedure is applied. Every strict call defines a barrier that clearly delineates the operations that must be performed before the application from the operations that must be performed after the application. For instance, in the monolithic versions of `mean-age` (see Chapter 2), strictness guarantees that each iteration of the loop will read the next record from the database, add the age of the current record to the running sum, and increment the running length. The operations of the next iteration cannot be performed until all the operations of the current one have been completed. In this way, strictness effectively manages the interleaving of operations from separate idioms.

In contrast, non-strict strategies (lazy and eager evaluation) decouple the time-based ordering of the argument computations from the time of the procedure application (see [HA87]). While non-strictness is a powerful and useful language feature ([Hug90, Tra88, Hal85, Mil87]), the lack of effective barriers thwarts efforts to reason about operational details like storage requirements and operation order. For example, both lazy and eager strategies give rise to insidious space leaks and introduce new complexities in programs with side-effects.

It is important to note that non-tail calls define *two* barriers. The *down barrier* is the barrier described above that delineates argument computation from the computation of the procedure body. The *up barrier* delineates the computation of the procedure body from the computation that uses the results of the call. In the tile diagrams of Chapter 4, a call boundary represents both the down and up barriers.

Up barriers can constrain the order of operations performed in the up phase of a computation. For example, consider the following procedure:

```
(define (sum&list-squares lst)
  (if (null? lst)
      (list 0 '())
      (mlet ((sq (square (car lst)))
             ((sum sqrs) (sum&list-squares (cdr lst))))
        (list (+ sq sum) (cons sq sqrs)))))
```

Given an input list, the procedure returns two results (packaged as a list): (1) the sum of the squares of the numbers in the input list and (2) a list of the squares of the numbers in the inputs list. In the down phase of a computation generated by `sum&list-squares`, each call is preceded by one occurrence each of `null?`, `car`, `cdr`, and `square`. In the up phase of the computation, each return is preceded by one occurrence each of `list`[1], `cons`, and `+`. Here the up barrier of the recursive call forces the list collection and summation idioms to proceed in lock step.

5.1.2 Distributing Strict Calls Loses Control

The main problem with decomposing a monolithic procedure into slivers is that this eliminates the barriers that provide control. Consider Figure 5.1, which depicts the computations associated with a monolithic program and the associated sliver program. Each dotted horizontal line represents a call boundary. The decomposition process splits each monolithic call boundary into one boundary for each sliver (the labels emphasize which boundaries match up).

The control supplied by the monolithic call are lost when it is distributed over the slivers. Whereas the down and up barrier of the monolithic call forces the idioms to work in lock step, the corresponding barriers of the slivers are only loosely coupled. Modulo dataflow dependencies, nothing prevents one sliver from racing ahead of its neighbors. For example, sliver S_1 may cross the down barriers of A, B, C, and D before S_2 has even crossed A. The lack of a shared barrier results in two problems:

1. *Unsynchronized Operations:* In the sliver computation, the corresponding operations of different idioms are no longer guaranteed to be synchronized. It is no longer possible to rely on properties that depend on the synchronized order.

[1]In actuality, `list` performs some `cons`es, but we'll treat it as a primitive for now.

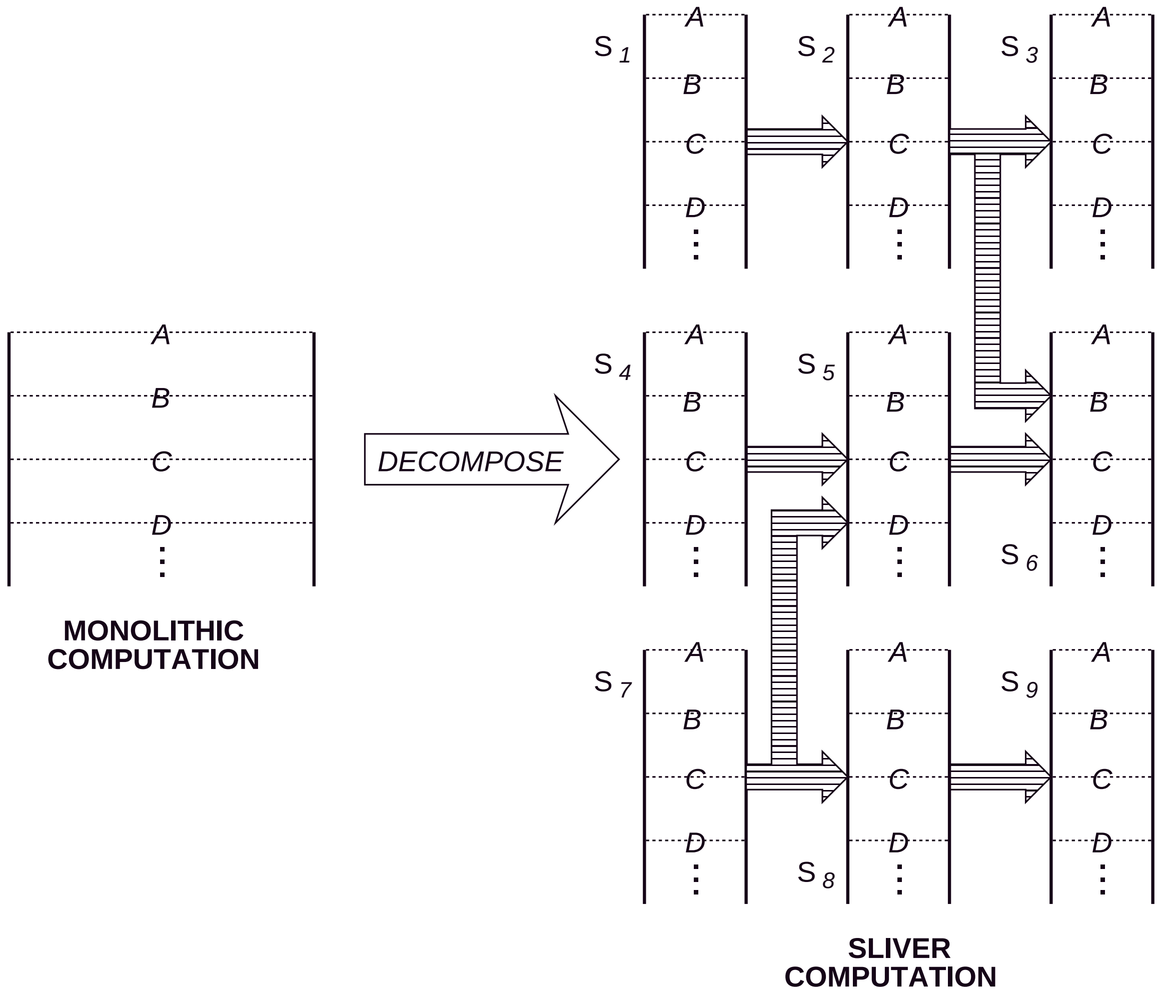


Figure 5.1: Decomposing a monolithic computation into a network of slivers.

2. *Buffering of Intermediate Results*: If a producing sliver races ahead of a consuming sliver, then the values produced by the producing sliver must be buffered somewhere until the consuming sliver consumes them. This buffering requires storage beyond that inherently implied by the computation.

In practice, the problem of buffering is far more serious than whether operations are synchronized. But the two problems are closely related. If the operations are not synchronized appropriately, then buffering will be required.

The space of possible operation schedules (i.e., orderings of operations) that are consistent with the sliver computation are generally much larger than the schedules consistent with the monolithic computation. Each of the SPS techniques studied in Chapter 3 picks a particular schedule from this larger set. For example, a strict aggregate approach chooses to perform all operations in one sliver before moving on to another one. None of the techniques is guaranteed to pick a schedule that is consistent with the monolithic computation. While the concurrent approaches (e.g., the concurrent channel technique and Hughes's `par/synch` technique) may pick a monolithic schedule, they are not guaranteed to.[2]

5.1.3 Simulating Strict Calls Regains Control

Intuitively, the lock step processing model guarantees desirable behavior for a sliver network by gluing the corresponding call boundaries of the slivers together so that they simulate the call boundaries of the monolithic call. Each sliver call boundary locally provides down and up barriers for the operations performed within the sliver. If the corresponding call boundaries were aligned so that all calls occurred at the same time and all returns occurred at the same time, then the sliver network would be forced to follow an operation schedule consistent with the monolithic computation. This approach clearly solves the problem of unsynchronized operations; and as long as the gluing process itself does not consume an unreasonable amount of storage, it solves the unwanted buffering problem as well.

An important wrinkle on the gluing idea concerns the handling of tail calls. Some of the sliver call boundaries may locally be tail calls. We want to guarantee that the gluing

[2] It's still possible to guarantee good space behavior without following one of the monolithic schedules.

process does not force a sliver to push stack when there is no pending work to be done. In particular, if all the corresponding call boundaries are tail calls, we want to guarantee that the entire network effectively performs a monolithic tail call. This behavior makes it possible to compose iterative computations out of iterative slivers, and has desirable consequences for general tree-recursive computations as well.

The notion of gluing together the corresponding call boundaries of networked slivers is very simple. The hard part is designing a mechanism that accomplishes it. We can argue from first principles the important properties that such a mechanism must possess:

- *Concurrency*: As Hughes has shown, any sequential evaluation strategy is insufficient for achieving desired space behavior in certain networks with fan-out [Hug83, Hug84]. The lock step model requires that the slivers are somehow executing concurrently.
- *Synchronization*: Concurrency *allows* the desired behavior but it doesn't necessarily *guarantee* it. Some form of synchronization is required to get the effect of gluing the call boundaries together. This form of synchronization must be more stringent than the synchronization associated with the other concurrent techniques we have studied. For the sliver computation diagram in Figure 5.1, for instance, both Hughes's technique and the concurrent channel technique allow sliver S_1 to be at the call labelled D while S_9 is at the call labelled A. The lock step model requires that no sliver can reach a call labelled B until all have passed through the call labelled A.
- *Non-strictness*: To simulate demand-driven evaluation in a monolithic computation, the lock step model requires that the values produced by a sliver are not computed unless they are actually needed by the consuming sliver. This implies that the model supports some form of laziness.

5.1.4 Lock Step Components

The lock step processing model is fine for simulating a single monolithic recursive procedure. But a typical computation is specified by many monolithic procedures. It is not appropriate for the slivers corresponding to one procedure to be acting in lock step with the slivers from another procedure. To deal with this situation, the model assumes that the slivers of a computation are partitioned into *lock step components*, where each lock step component consists of slivers that are intended to proceed in lock step.

Lock step components can be wired together to yield a loosely coupled network of tightly coupled parts. Such a network is depicted in Figure 5.2. The figure contains four lock step components, which are denoted by dotted outlines. Each component consists of slivers that are connected by thick cables, while the components themselves are connected by thin wires. Henceforth, a cable connecting two slivers shall be taken as a declaration that they are in the same lock step component. All other communication is accomplished by wires. As indicated by the figure, non-sliver computational devices (`X`, `Y`, and `Z`) may be attached to the wires.

5.1.5 The Details

The remaining sections of this chapter describe the details by which the lock step model outlined above can be achieved. The presentation consists of the following parts:

- *Sliver Decomposition*: Based on the ideas introduced in Chapter 4, this section describes how to decompose a monolithic computation into a network of slivers.

- *The Structure of Synchronized Lazy Aggregates*: This section motivates the requirements for a data structure that supports the lock step processing model. Most importantly, it introduces a novel synchronization technology, the *synchron*, that permits call boundaries to be glued together.

- *Slivers Revisited*: Synchronized lazy aggregates provide the raw materials for gluing call boundaries together, but the slivers are responsible for hooking everything up in the right way. This section describes the details of the gluing process.

- *Filtering*: One of the trickiest aspects of synchronized lazy aggregates is handling filtering. This section explains how it's done.

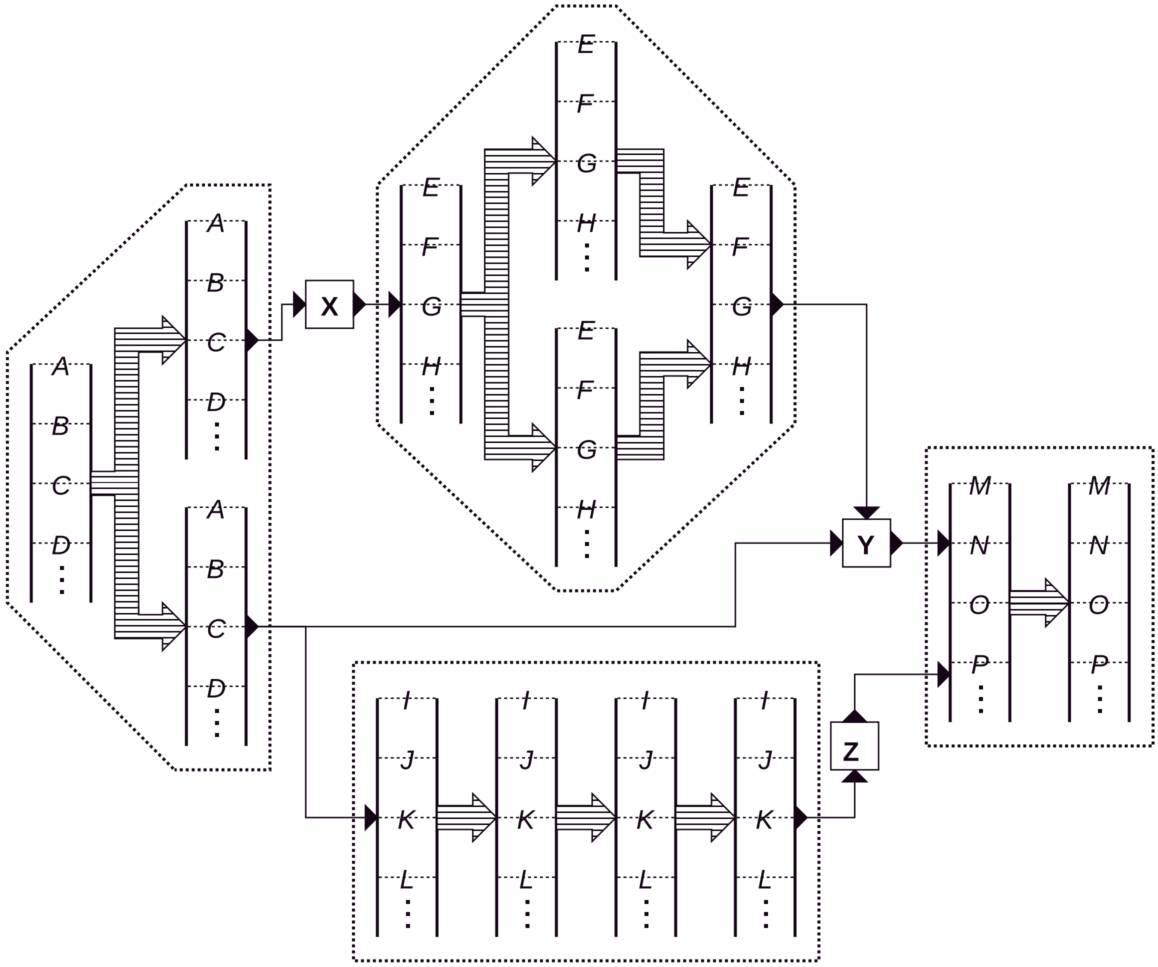


Figure 5.2: A network consisting of four lock step components. Slivers connected by thick cables form a lock step component. Lock step components are connected by thin wires.

5.2 Sliver Decomposition

Chapter 2 introduced sliver decomposition as a way of decomposing monolithic computations into slivers. Here, I use the shape concepts developed in Chapter 4 to describe sliver decomposition more precisely. As in Chapter 4, I will limit the discussion to unitilable monolithic recursions.

Without the synchronization supplied by synchronized lazy aggregates, it will not be possible to guarantee lock step behavior for the result of the decomposition. This section describes the first in a series of approximations to the final model.

5.2.1 Linear Subtiles

Figure 5.3 suggests a simple strategy for decomposing a unitilable monolithic computation into slivers: break the tile for the monolithic computation into communicating fragments called *subtiles*, and then replicate each subtile to reflect the structure of the monolithic computation. Although the figure shows a linear computation whose slivers communicate

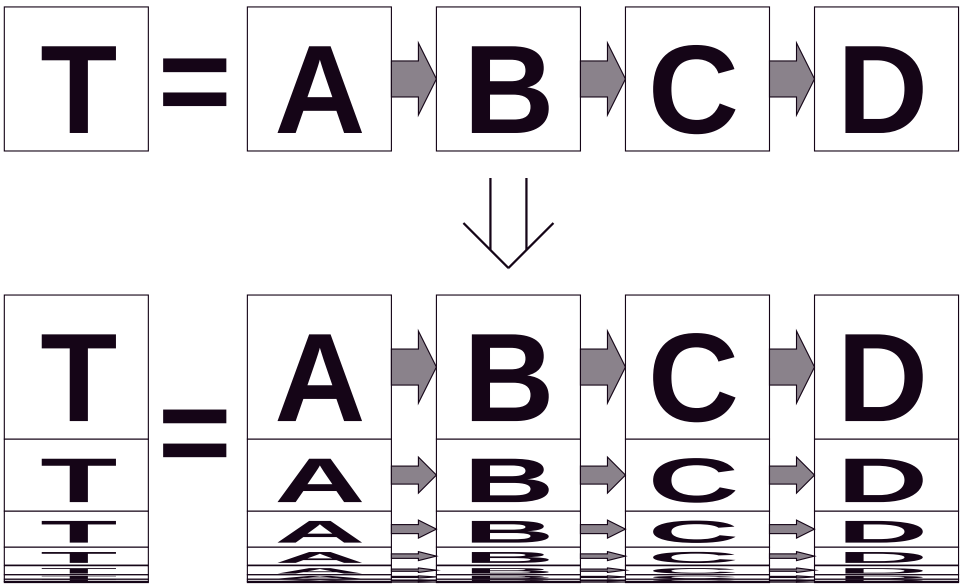
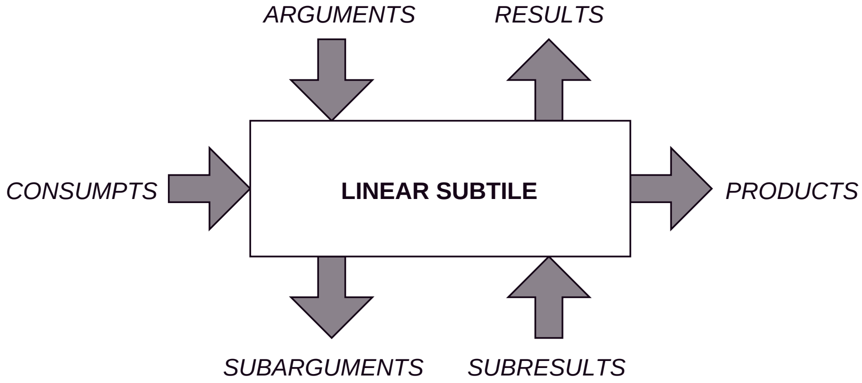


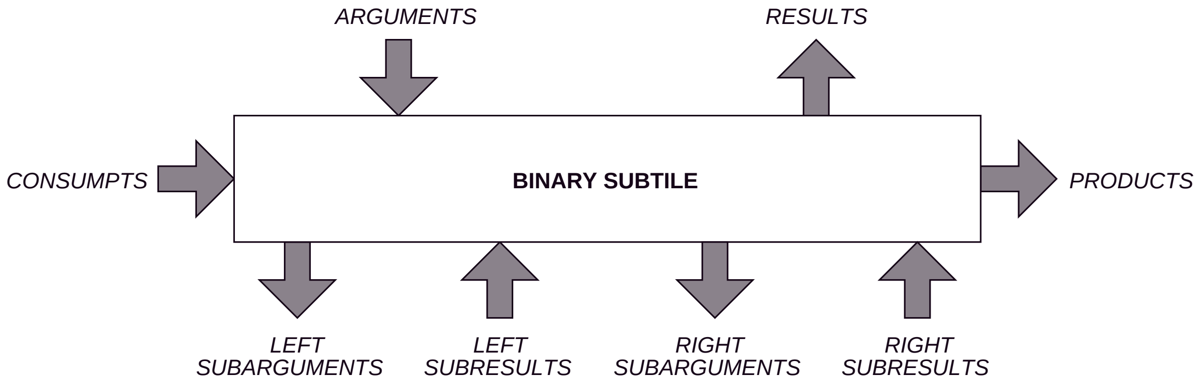
Figure 5.3: A simple strategy for sliver decomposition. If a tile (**T**) can be decomposed into communicating subtiles (**A**, **B**, **C**, and **D**), then the computation corresponding to the replicated tiles can be decomposed into the slivers corresponding to the replicated subtiles.

in a straight line, the strategy applies to tree computations and sliver networks that exhibit fan-in and fan-out.

A subtile is a generalization of a tile that can communicate horizontally with other subtiles in addition to communicating vertically with copies of itself (see Figure 5.4). The



(a) Linear subtile.



(b) Binary subtile.

Figure 5.4: Subtile interfaces.

horizontal inputs are called *consumpts* and the horizontal outputs are called *products*. As indicated by Figure 5.3, each subtile instance within a trellis can communicate horizontally only with instances of other subtiles at the same trellis location. A tile can be viewed as special kind of subtile whose consumpts and products are both empty.

Figure 5.5 illustrates a first cut at using this strategy to modularize the recursive sum-of-squared-evens tile from Figure 4.6. The tile is decomposed into four subtiles that correspond

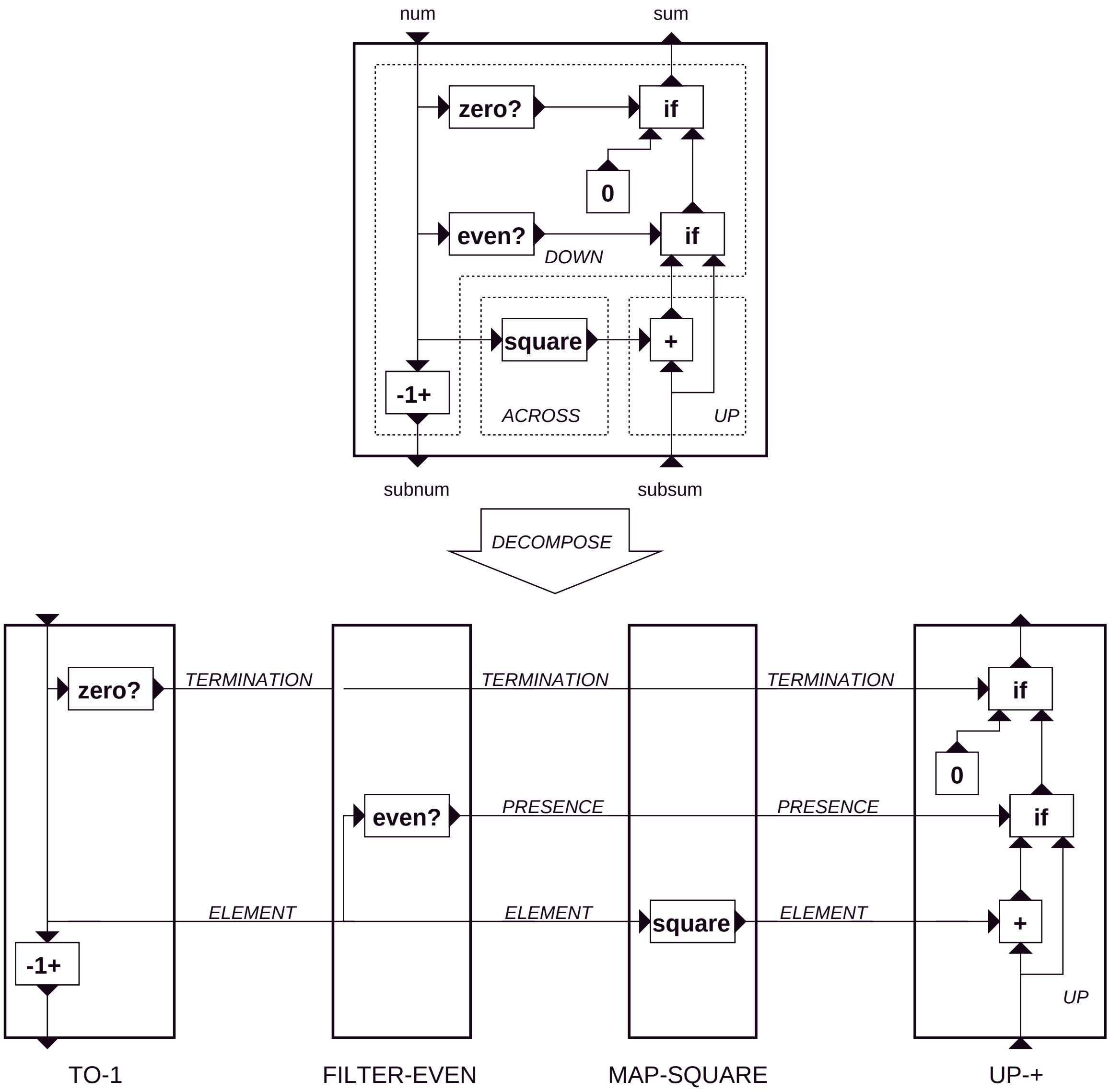


Figure 5.5: A naive decomposition of the sum-of-squared-evens tile into four subtiles.

to common programming idioms:

1. `TO-1` generates a sequence of integers from a given integer down to 1 (inclusive).

2. `FILTER-EVEN` is a sequence filter that passes only the even numbers from its input to its output.

3. `MAP-SQUARES` squares each of the elements in the input sequence.

4. `UP-+` performs a recursive sum accumulation on a sequence.

The subtiles communicate information horizontally via three classes of wires:

1. *Termination wires* transmit a boolean that indicates whether or not the end of the sequence has been reached.

2. *Presence wires* transmit a boolean that indicates whether the associated element has been passed by a filter.

3. *Element wires* transmit the current element of the sequence.

Unfortunately, the decomposition depicted in Figure 5.5 suffers from two very important problems:

1. *Non-standard interfaces*: Reusability is hindered because the components don't all share the same interface. In particular, some subtiles handle presence wires, while others do not. For instance, `FILTER-EVEN` produces a presence wire even though it does not consume one. Two such filters cannot be cascaded — a disaster for composability. Naive decompositions for other tiles presented earlier can easily lead to subtiles in which some of the three wires are absent or duplicated.

2. *Inadequate control*: Operational faithfulness is threatened by the inadequate specification of when subtile operations or subcalls are performed. For example, the simple operational rules outlined previously dictate that, within the tile, the `-1+` is performed only after the subresult has been requested. However, in the decomposed version, different subtiles have different subcall boundaries. How does the `TO-1` tile know when

> to perform `-1+`? Similarly, the `MAP-SQUARE` subtile does not explicitly specify that the `square` operation is performed only on even integers; yet this is clear in the tile. And none of `TO-1`, `FILTER-EVEN`, and `MAP-SQUARE` indicate *when* a subcall should be initiated. Since slivers are supposed to mimic the operational behavior of a monolithic computation, this underspecification of control is disconcerting.

The improved tile decomposition in Figure 5.6 addresses some of these concerns. First, all of the subtiles share a standard interface in which each sequence element is represented by a triple of termination, presence, and element wires. The presence wire is manipulated by all subtiles in a composable way: the generator initializes the presence wire to a true value (`#t`); the filter combines it with local filtering information; the mapper passes it along untouched; and the accumulator uses it to control accumulation. The subtiles also maintain the invariant that when the presence wire carries the false value, the element wire carries a distinguished *gap* value (written `#g`). The gap value indicates a position in the sequence where an element has been filtered out.

Second, control details are much more explicit in the subtiles of Figure 5.6 than those of Figure 5.5. Each subtile has its own copy of the termination control assembly from the tile's control arm. The generator, filter, and mapper are all assumed to ultimately return the boolean truth value (`#t`) just so that the request for this value can be used as a means of specifying local control. For example, the `-1+` operation in the `TO-1` tile will only be performed after its local subresult has been requested. The fact that `MAP-SQUARE`'s `square` operation is guarded by a test of the presence wire is another example of control being made explicit.

The more sophisticated decomposition still leaves some important control problems unsolved. The subcall boundary of the whole tile effectively synchronizes all subresult requests with all subargument evaluations. When the subcall boundary is distributed across subtiles, one sliver can easily race ahead of another. For example, nothing prevents the sophisticated `TO-1` sliver from merrily generating all the numbers in the sequence before any of the other slivers have processed the first one. This would imply the need for storage buffers between the components, which is exactly the kind of behavior the lock step model is supposed to avoid. We will see shortly how synchronized lazy aggregates solve this problem. Until then,

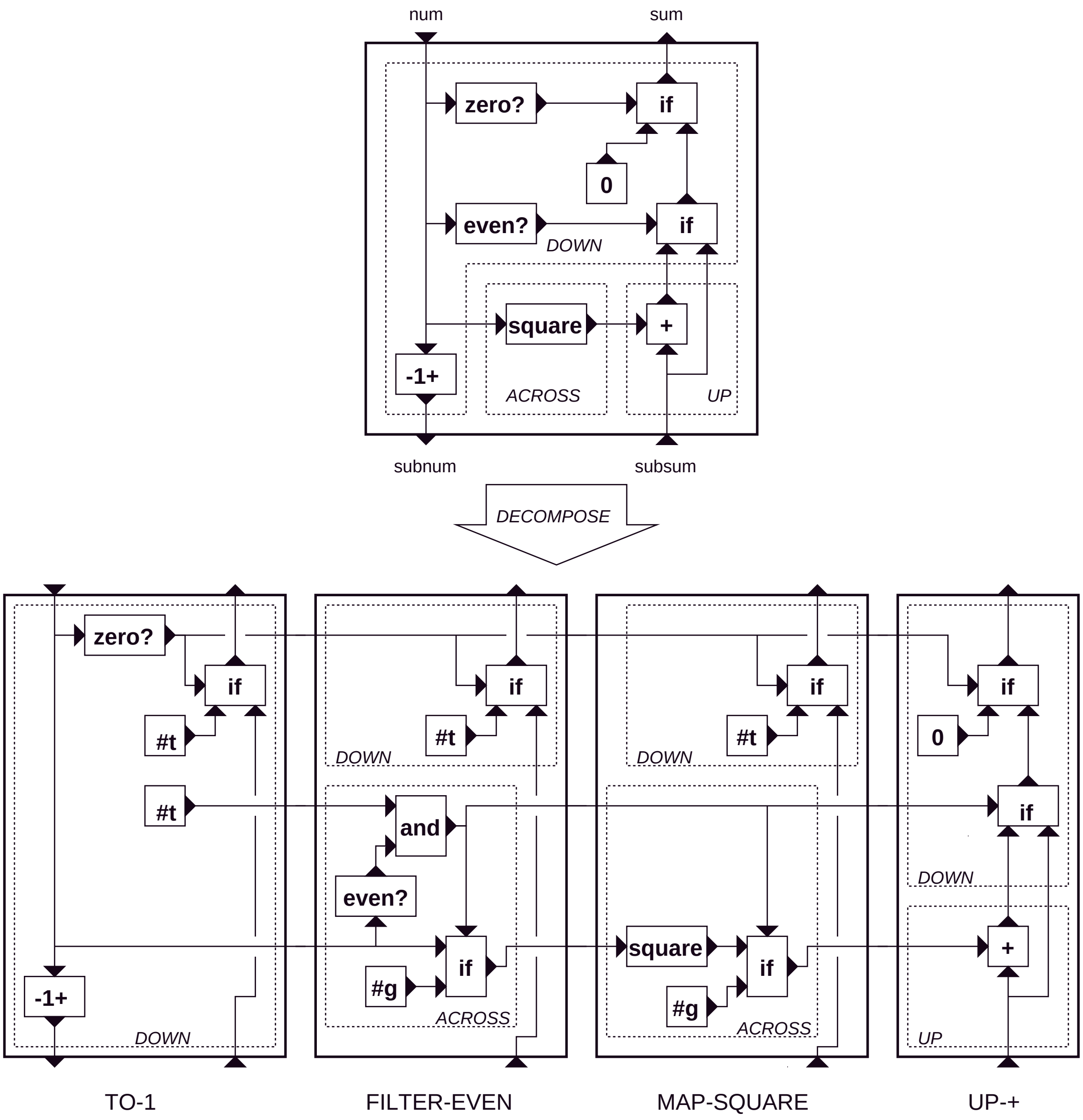


Figure 5.6: A sophisticated decomposition of the sum-of-squared-evens tile into four subtiles.

we will assume that connected slivers magically compute in a synchronous manner.

Another drawback of the sophisticated decomposition is that the four subtiles obviously do more total work than the single tile. Some examples:

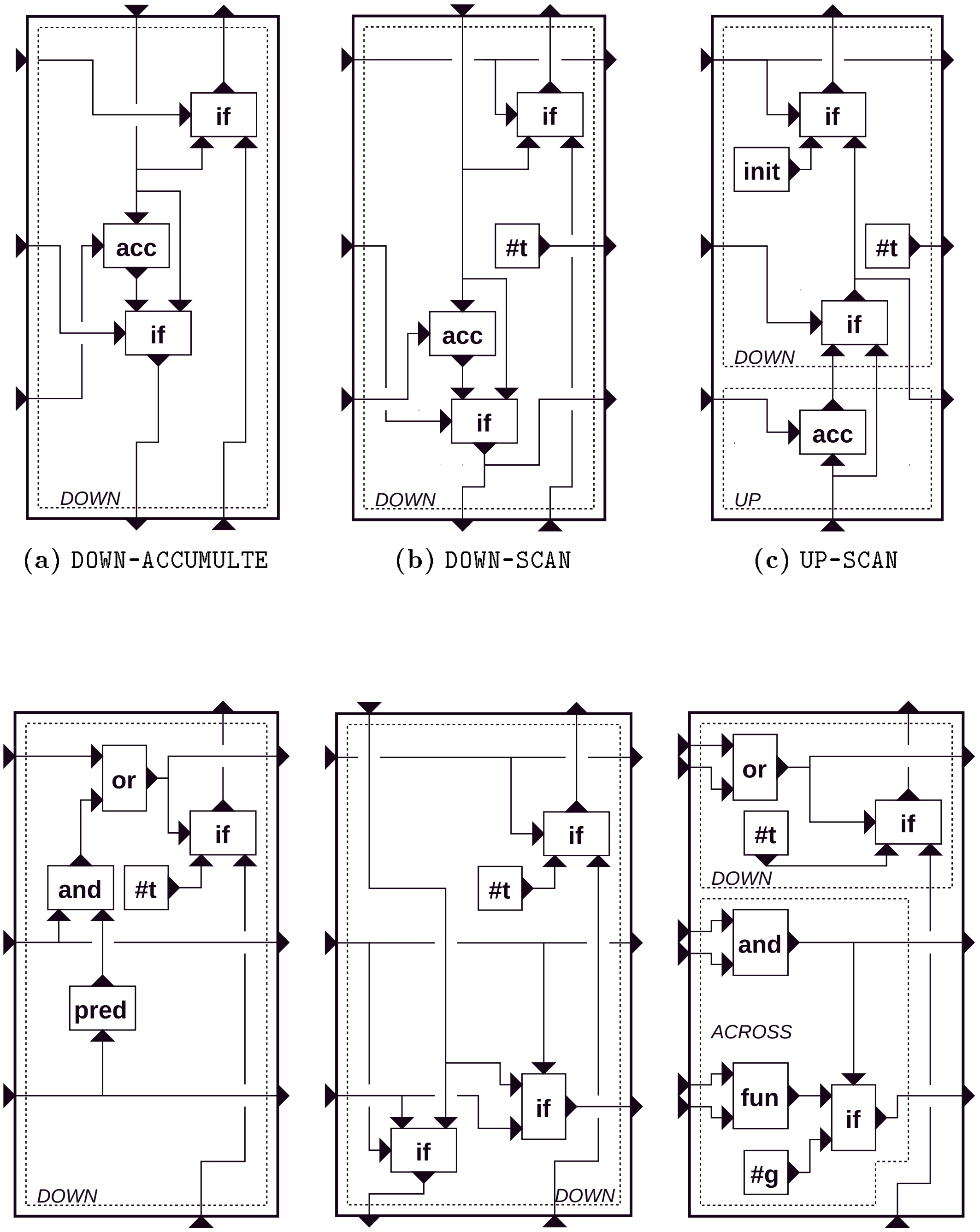
- The termination test is performed once by the tile, but four times by the subtiles.
- Manipulating presence wires in a standard way requires extra `if` and `and` operations.
- The communication and synchronization between subtiles (introduced later) will undoubtedly require additional overhead as well.

It is possible that executing the slivers on multiple physical processors could reduce these overheads. But this is unlikely[3] and, more relevant, unimportant. Most any attempt to provide standard interface within a given system is bound to result in overheads. These are often justified by a significant gain in simplicity. The kinds of overhead enumerated above can be insignificant when compared to the mental overhead of having to express computations in a non-modular way. The *real* benefit of sliver decompositions is that they suggest new ways of analyzing and synthesizing computations.

A wide range of useful subtiles can be designed with the standard three-wire-per-element interface introduced above. Figure 5.7 illustrates some more linear subtiles. Subtile **(a)** is an iterative accumulator; if the `UP-+` subtile of Figure 5.6 were replaced by an instance of this down accumulator, the network would correspond to an iterative sum-of-squared evens rather than a recursive one. Subtiles **(b)** and **(c)** are *scanners* that emit as products the intermediate accumulated values in a *down* or *up* accumulation.[4] Note that a scanner product is always present even if the corresponding consumpt was not present. Subtile **(d)** is a *truncater* that terminates a sequence as soon as `pred` is true of an element; the (short-circuit) `and` prevents `pred` from being applied to a gap. Subtile **(e)** is a *shifter* that moves non-gap elements down to the next non-gap rung in a linear trellis. Subtile **(f)** maps a

[3]Since the slivers do so little computation between communication and synchronization events, parallel execution of slivers is likely to offer few practical performance benefits.

[4]There are two very different meanings of "scan" in the literature on the signal processing style of programming. In the data parallel literature, "scan" refers to a partial accumulator [Ble90, Sab88, GJSO92]. In Waters's series package, though, "scan" refers to a kind of generator [Wat90]. I adopt the former meaning here.



(a) DOWN-ACCUMULTE (b) DOWN-SCAN (c) UP-SCAN

(d) TRUNCATE (e) SHIFT (f) MAP2

Figure 5.7: A gallery of linear subtiles.

function `fun` over the elements of two consumpts to give a single product. The termination wires of `MAP2` are joined with an `or` so that the product terminates as soon as one of the input terminates. The presence wires of `MAP2` are joined with `and` so that the product is only present (i.e., the function is only applied) if both consumpts are present.

The visual complexity of subtile diagrams has limited this discussion to very simple sliver networks. Examples of more complex networks specified in a textual form appear in Chapter 6.

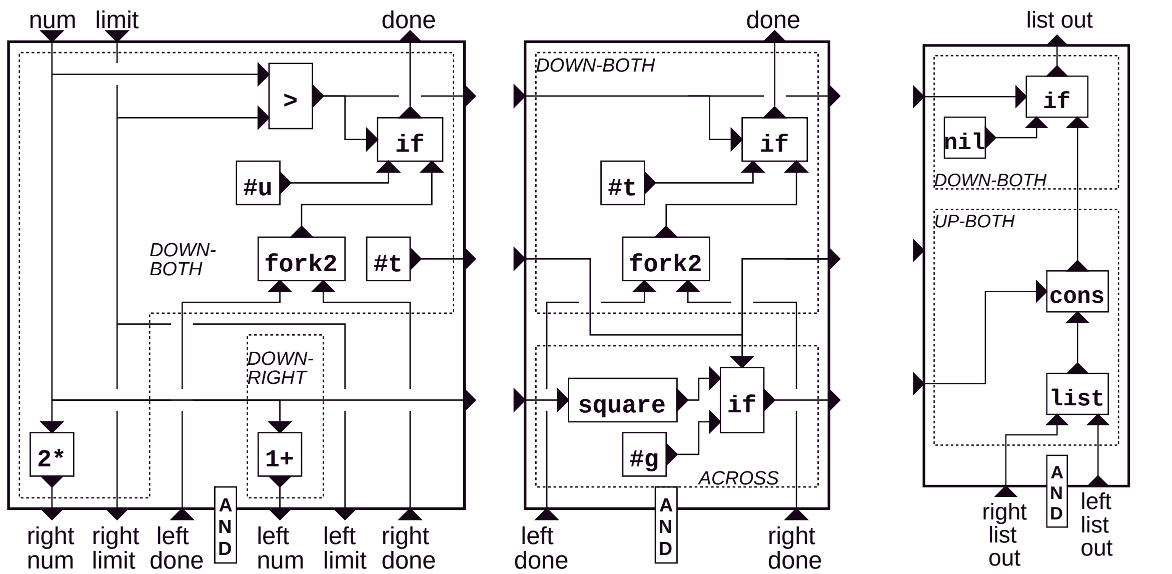
5.2.2 Binary Subtiles

The kinds of subtile decompositions introduced above extend naturally to tree-shaped computations. Figure 5.8 shows some sample binary subtiles. As in the linear case, each tree node is represented by a triple of termination, presence, and element wires. Conceptually, the `BREADTH-INDEX` generator (subtile **(a)**) can be combined pairwise with the four accumulating subtiles (**(c)** – **(f)**) to yield the tiles in Figure 4.19. Accumulating the squares of the elements of a breadth index tree could be accomplished by inserting the `BINARY-MAP-SQUARE` tile between generator and accumulator.

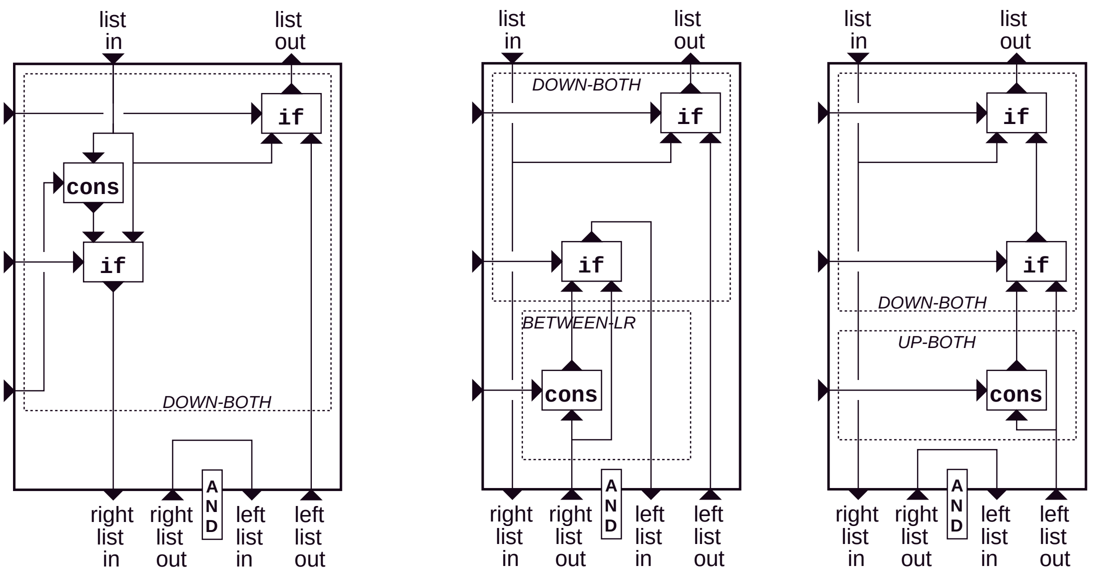
Perhaps the biggest surprise in Figure 5.8 is the presence of the non-strict `fork2` node in both `BREADTH-INDEX` and `BINARY-MAP-SQUARE`. Neither of these subtiles naturally accumulates any meaningful value. The duty of the `done` port in both cases is not to return a value but to propagate requests to the subcalls. In order to maintain the essential parallel, non-accumulating character of these subtiles, it is imperative to propagate requests in a way that both (1) does not specify the order of the subcalls and (2) does not leave behind a pending operation to be performed. Single-threading the request through the two subcalls would satisfy (2) but not (1). On the other hand, the eager, parallel behavior of `fork2` fits the bill perfectly. So while we will not permit whole tiles to use `fork2`, we will use `fork2` in a restricted way to represent subtiles with a *binary down* character.[5]

The only other surprise is `COLLECT-TREE`'s treatment of gaps. Note that the presence wire is never checked! When performing a *parallel up* accumulation on a tree with gaps,

[5]Conceptually, when subtiles are glued together to form tiles, there will necessarily be some subtile with an accumulation component that renders the `fork2`s unnecessary. So no tile composed out of subtiles requires a `fork2`.



(a) BREADTH-INDEX (b) BINARY-MAP-SQUARE (c) COLLECT-TREE



(d) BINARY-PRE-CONS (e) BINARY-IN-CONS (f) BINARY-POST-CONS

Figure 5.8: Sample binary subtiles.

it is generally necessary to specify *two* accumulators: one for the case where the current element is present, and one for the case where it is not. `COLLECT-TREE` makes the simplifying assumption that it's alright to include a gap value in the returned tree. But if a sum accumulation were being performed instead, the innards of the sliver would be more complex.

5.2.3 Subtile Shapes

Subtiles can be classified by a notion of shape similar to that for tiles. As with tiles, the operations of subtiles can naturally be partitioned into shards according to their relationship with subcall initiation and return. For example, such shards are labelled in the subtiles of Figures 5.6 – 5.8.[6]

The subtile shards stand out from tile shards in several ways:

- In Figure 5.6, the *down* shards of `FILTER-EVEN`, `MAP-SQUARE`, and `UP-+` consist purely of control arms. In contrast, the *down* shards from all the linear tile examples presented thus far have all been connected to the subcall boundary via a subargument wire. Similar remarks hold for the *down-both* shards of `BINARY-MAP-SQUARE` and `COLLECT-TREE` in Figure 5.8.

- Subtiles can have a shape different from that of the tile from which they are derived. For example, the `TO-1` subtile in Figure 5.6 has a fundamentally *down* shape even though the tile as a whole has an *up* shape. This captures the intuition that the generator fragment of the tile works in a fundamentally iterative way. Similarly, the `BREADTH-INDEX` subtile in Figure 5.8 has a fundamentally *binary down* shape even though none of the tiles in Figure 4.19 (page 154) has *binary down* shape.

- Due to horizontal communication, subtiles support patterns that are not possible for whole tiles. For example, `FILTER-EVEN` and `MAP-SQUARE` in Figure 5.6 each has nontrivial *down* and *across* shards but a trivial *up* shard. This pattern is not possible in a tile because the results of the *across* shard would necessarily be consumed by

[6]For simplicity, trivial shards have been omitted in these and subsequent figures.

a non-trivial *up* shard. In subtiles, an *across* shard can find an alternate outlet as subtile products. We shall use the term *across shape* to refer to a linear subtile that consists only of an *across* shard in addition to a simple termination-controlling *down* shard. Similarly, `BINARY-MAP-SQUARE` will be said to have a *binary across* shape.

Except for the *across* shapes mentioned above, subtile shapes are determined in a manner similar to tile shapes. For example, the shape definitions in Table 4.2 (page 155) hold for binary shapes.

An interesting feature of *across* subtiles is that their operational behavior is context dependent. Consider the three different uses of the `MAP-SQUARE` subtile in Figure 5.9:

1. In network **(a)**, `MAP-SQUARE` feeds a *down* accumulator that forces the squaring operation to be performed before subcall initiation.

2. In network **(b)**, `MAP-SQUARE`'s position between a *down* generator and an *up* accumulator leaves the squaring operation unconstrained with respect to the subcall.

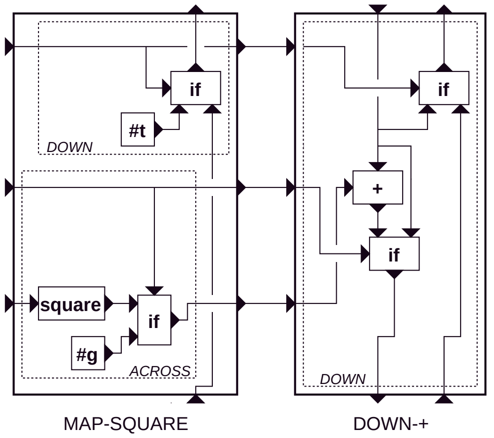
3. In network **(c)**, `MAP-SQUARE` is wedged between an *up* scanner and an *up* accumulator, which force the squaring operation to occur after the subcall returns.[7]

These examples illustrate that an *across* subtile operation does not necessarily act like an *across* operation in a network of subtiles; its behavioral fate is determined by the surrounding context. In contrast, *down* and *up* subtile operations are unaffected by context. So while cascading two linear subtiles always yields another linear subtile, it may require the relabelling of *across* shards.

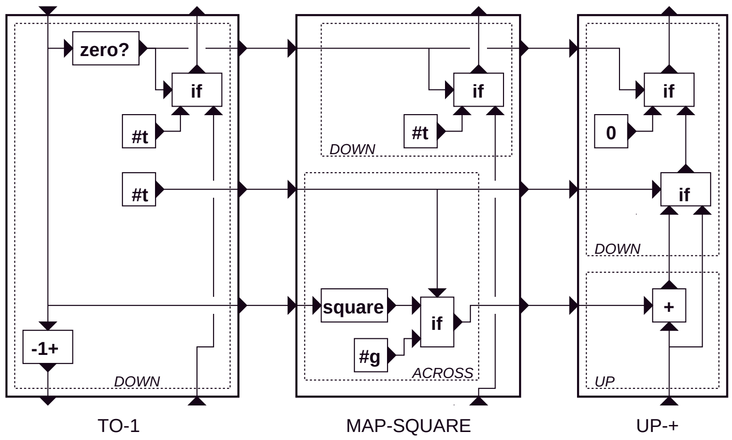
5.2.4 Sliver Computations

Just as replicating a tile throughout a trellis gives rise to a monolithic computation, replicating a subtile throughout a trellis conceptually generates a *sliver computation*. Like monolithic computations, sliver computations have a dynamic shape property determined by the time structure of their call events.

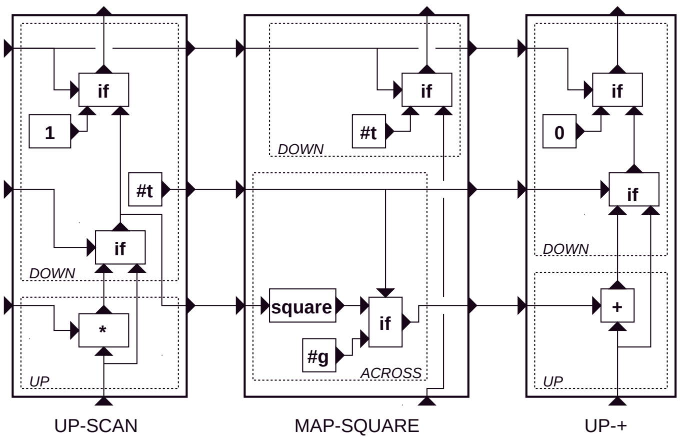
[7] There is actually a nasty technical problem lurking in network **(c)** that will be discussed in Section 5.5.



(a) `MAP-SQUARE`'s *across* shard forced down.



(b) `MAP-SQUARE`'s *across* shard unconstrained.



(c) `MAP-SQUARE`'s *across* shard forced up.

Figure 5.9: Examples showing the context-dependent nature of *across* operations within a subtile.

Due to consumpts and products, sliver computations within the same lock step component can be viewed as consuming and producing horizontally transmitted values. Because of the demand driven nature of the components out of which they are built, no product is computed unless it is actually needed by a consuming sliver. Thus, this version of the sliver decomposition technique is operationally faithful in this detail.

5.2.5 Subtile Compatibility

While subtiles share a standard interface, not all combinations of subtiles make sense. First of all, there are some obvious structural constraints:

- Connected subtiles must have the same number of subcalls. It doesn't make sense to mix linear and binary subtiles.[8]

- Connections are only allowed between result/argument ports and between product/consumpt ports. It is illegal to wire a result port to a consumpt port or a product port to an argument port.

- A product/consumpt connection is only legal if the termination/presence/element ports are correspondingly wired. That is, a triple of termination, presence, and element ports is treated as a single connection point.

All of these are local constraints that can be simply checked between any pair of subtiles.

More subtly, there are some important non-local constraints. Consider the subtile network in Figure 5.10, in which an up-multiplying scanner feeds an down-summing accumulator through a square mapper. According to the operational model, the `+` must be performed before subcall initiation in `DOWN-+`, and the `*` can only be performed after subcall return in `UP-*`. But the subcall boundaries of connected slivers are conceptually glued together to form a single boundary. So the shaded path indicates a dependency circularity in the subtile network that will cause deadlock in any computation based on this network. This circularity is non-local because it cannot be discovered by considering any connected pair of subtiles.

[8] Actually, this isn't quite true. It is possible to imagine subtiles that "convert" between linear and tree slivers.

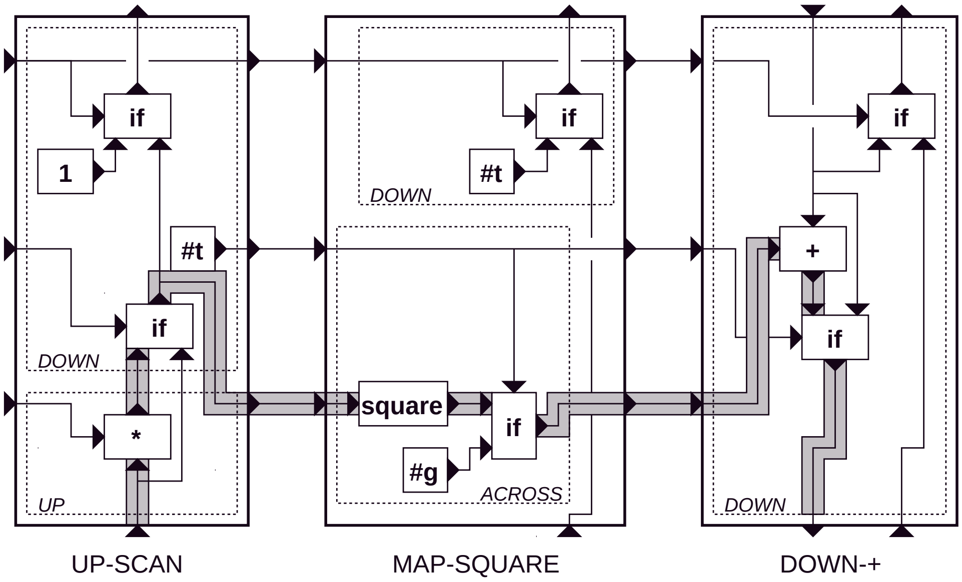


Figure 5.10: This network of subtiles is non-sensical because the corresponding tile is guaranteed to deadlock.

Another way to view the deadlock problem of Figure 5.10 is in terms of *up* and *down* phases. Recall that every unitilable linear computation has a single *down* phase followed (optionally) by a single *up* phase. The problem with any computation based on the sample network is that it attempts to force a second *down* phase to occur after the *up* phase. Intuitively, this cannot be done in a single recursive pass; it requires two recursive passes communicating via an aggregate data structure. By design, lock step components model only computations with one recursive pass, so no single subtile network can represent the intended computation. (However, it is possible to encode the computation as two lock step components.)

There are two basic approaches for detecting deadlock within a subtile network:

1. *Dynamic deadlock detection*: In a dynamic approach, deadlock is only detected during the execution of the computation generated by a subtile network. Deadlock is detected by the execution engine when no progress can be made on a computation that has not terminated.

2. *Static deadlock detection*: In a static approach, a subtile network is analyzed for potential deadlocks before it is executed. Deadlock is detected by the analyzer when it discovers a potential dependency circularity.

These approaches exhibit the classic tradeoffs found in other dynamic/static dichotomies (e.g. dynamic vs. static type-checking). Static deadlock detection is helpful for finding logical errors as early as possible. However, it requires the conceptual overhead of developing a theory of deadlock and inventing a sound deadlock detection algorithm. Moreover, deadlock is an undecidable property (see below), so any sound static deadlock analysis will conservatively ascribe deadlock to some deadlock-free computations.

In contrast, dynamic deadlock detection offers simplicity and expressibility. The existing operational model is already powerful enough to detect deadlock. And by delaying deadlock detection as long as possible, the dynamic approach finds no spurious deadlocks. On the other hand, this approach provides no guarantees that a program is deadlock-free. And while a system can provide debugging information on a deadlocked state, it can be difficult to trace the deadlock back to its source.

Since shapes encode dependency information, it should be possible to develop a form of static deadlock detection based on subtile shapes. For example, in the case of linear subtiles, we expect that an *up* tile feeding a *down* tile can cause deadlock, but we know that it is always safe for a *down* subtile to feed another *down* subtile. Similarly, with binary subtiles, we expect that left-to-right and right-to-left subtiles never mix (because this would imply incompatible traversals of the computation tree). However, *binary down* subtiles should be able to precede any binary subtile, while *binary up* subtiles should be able to follow any binary subtile. We shall refer to this shape-based approach for deadlock detection as *shape checking*, in analogy to type checking.

Although the possibility of shape checking is alluring, I have not yet developed an elegant formulation for it. While the combination rules mentioned in the previous paragraph are helpful heuristics, the notion of subtile shape defined earlier is too coarse for handling the nuances of deadlock in a reasonable way. Presumably, something more like the circularity detection analysis of attribute grammars [DJL88] is required here, but I will not explore this avenue. For simplicity, I will assume dynamic deadlock detection throughout the rest of this document.

I conclude this section with the sketch of a proof that deadlock is undecidable. Consider the simple computation diagram sketched in Figure 5.11. It contains a single-argument procedure `F`, `not` (the boolean negation operator), and two copies of an `EITHER-SCAN-+` sliver. The `EITHER-SCAN-+` sliver is designed to behave like `DOWN-SCAN-+` if its argument is

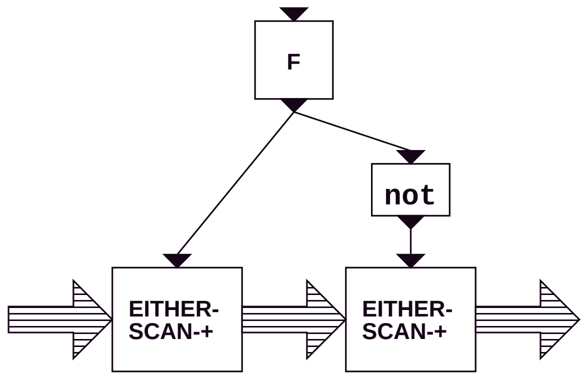


Figure 5.11: A network used to illustrate the undecidability of deadlock detection.

true, but behaves like `UP-SCAN-+` if its argument is false. Assuming that `F` is deadlock-free, the network can only exhibit a deadlock when the output of `F` is false. So the question of whether the network exhibits deadlock is reducible to the question of whether a given procedure is identically true for all inputs. The latter is obviously undecidable in general.

5.3 The Structure of Synchronized Lazy Aggregates

Subtiles are an abstract solution to the problem of partitioning a monolithic recursive computation into reusable fragments. There are many concrete questions they do not address:

- How is the computational pattern specified by a subtile replicated to generate a sliver computation?
- How are the products of one subtile communicated as consumpts to another subtile?

- What synchronization mechanism glues the corresponding call boundaries of connected subtiles together?

All of these issues are resolved by *synchronized lazy aggregates* (abbreviated as *slags*) a novel class of lazy data structures that carry explicit synchronization information:

- Since it represents a fragment of a monolithic recursive procedure, a sliver is naturally implemented as a recursive procedure that manipulates slags. The usual recursion mechanism accounts for subtile replication.

- As an aggregate data structure, a slag is well-equipped to communicate termination, presence, and element information from one sliver to another. The lazy nature of slags guarantees that communication happens in a demand-driven manner that faithfully models the operational behavior of a corresponding monolithic procedure.

- The synchronization information carried by a slag manages the lock step computation of connected slivers. Every sliver that abides by the rules of a synchronization contract is guaranteed not to race ahead or lag behind its compatriots.

5.3.1 Overview

We know from Chapter 3 that aggregate data is an elegant and easy-to-use technique for communicating values between sliver-like program components. We will adapt the technique to solve the problem of gluing together the corresponding call boundaries of all the slivers in a lock step component. The call boundaries influence the aggregate data approach in two fundamental ways:

1. *Synchronization*: In order for slivers to engage in lock step processing, the aggregate data structures connecting them must transmit some sort of synchronization tokens that represent the down and up barriers of the desired composite call boundary. I will call these tokens *synchrons*.

2. *Laziness:* Elements computed above a call boundary will have to be successfully communicated between slivers before elements below the call boundary have been

computed. This means that an aggregate connecting two slivers is generally only partially determined at a given point in time. A form of laziness will be used to represent the time-unfolding nature of the aggregate.

We will use the term *synchronized lazy aggregate* (abbreviated *slag*), to refer to an aggregate structure that addresses these two issues. Slags are a particular implementation of the cables that appear in sliver diagrams.

Intuitively, a slag is a data structure that represents the values of a program variable over time. Waters uses the term *temporal abstraction* [Wat78] to refer to this notion. In fact, his series data structure [Wat90, Wat91] is a particular instance of the more general synchronized lazy aggregates. Whereas a series represents the successive values of the state variable of a loop, synchronized lazy aggregates can represent the conceptual tree of values taken on by an identifier within an arbitrary recursive procedure. Another way to say this is that series corresponds to a register while synchronized lazy aggregates correspond to a register plus a stack.

Slags can fruitfully 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. (However, unlike many other synchronization models (e.g. [Hoa85, Mil89]), slags decouple communication and synchronization.)

5.3.2 Synquences and Syndrites

For simplicity, we will focus on two particular kinds of slags: *synquences* and *syndrites*. A synquence (*syn*chronized se*quence*) is a synchronized lazy list while a syndrite (*syn*chronized den*drite*) is a synchronized lazy tree. Synquences represent the cables between linear slivers, while syndrites represent the cables between tree-shaped slivers. While trees generally subsume lists as a special case, we distinguish the two structures because synchronized lists have important properties that not shared by synchronized trees. In particular, synchronized lists permit forms of iteration and filtering that are not supported by synchronized trees.

Intuitively, a synquence is a linear chain of value-bearing nodes linked together by pointers annotated with synchronization information. Figure 5.12 compares the structure of a traditional list to the structure of a synquence. In a traditional list **(a)**, the component values (`2`, `3`, and `5`) hang off of a chain of *skeletal nodes* consisting of three pairs and a `nil`. The synquence structure **(b)** is similar, except that each pointer to a skeletal node in **(a)** has been replaced by a compound structure holding onto two synchrons: a *down synchron* representing the down barrier of a call boundary, and an *up synchron* representing the up barrier of a call boundary. Structure **(c)** is an abbreviation of the synquence **(b)** that emphasizes how the synchronization information has essentially annotated the pointers of the list structure. A *barrion* stands for the pair of down and up synchrons.

A pointer annotated with a barrion will be called a *syn*chronized poin*ter*, or *synter* for short. The term "synquence" refers not to a skeletal node, but to a synter that points to a skeletal node. Synters only connect the skeletal nodes that form the backbone of the aggregate; component values hanging off of the backbone are held by unannotated pointers. A valueless `nil` node terminates a finite synquence; an unterminated synquence is conceptually infinite in length.

In a synquence, the structure between two synters encodes the information in one triple of termination, presence, and element wires. The termination boolean is represented by whether or not the skeletal node is `nil`. The presence boolean and element are assumed to be encoded in the value held by a non-`nil` node.

The structure of syndrites is similar to the structure of synquences (see Figure 5.13). The two main differences are:

1. Syndrite nodes are connected by synters to multiple children rather than just one.

2. A finite branch of a syndrite is terminated by a `leaf node` that has a value but zero children. In contrast, the `nil` node terminating a synquence carries no value.

At first glance, representing leaves by value-bearing nodes may seem at odds with the termination/presence/element model. However, experience suggests that in a large percentage of tree-shaped computations, subtiles transmit/receive elements when the corresponding termination boolean is true. The essence of termination in tree computations is not a value-

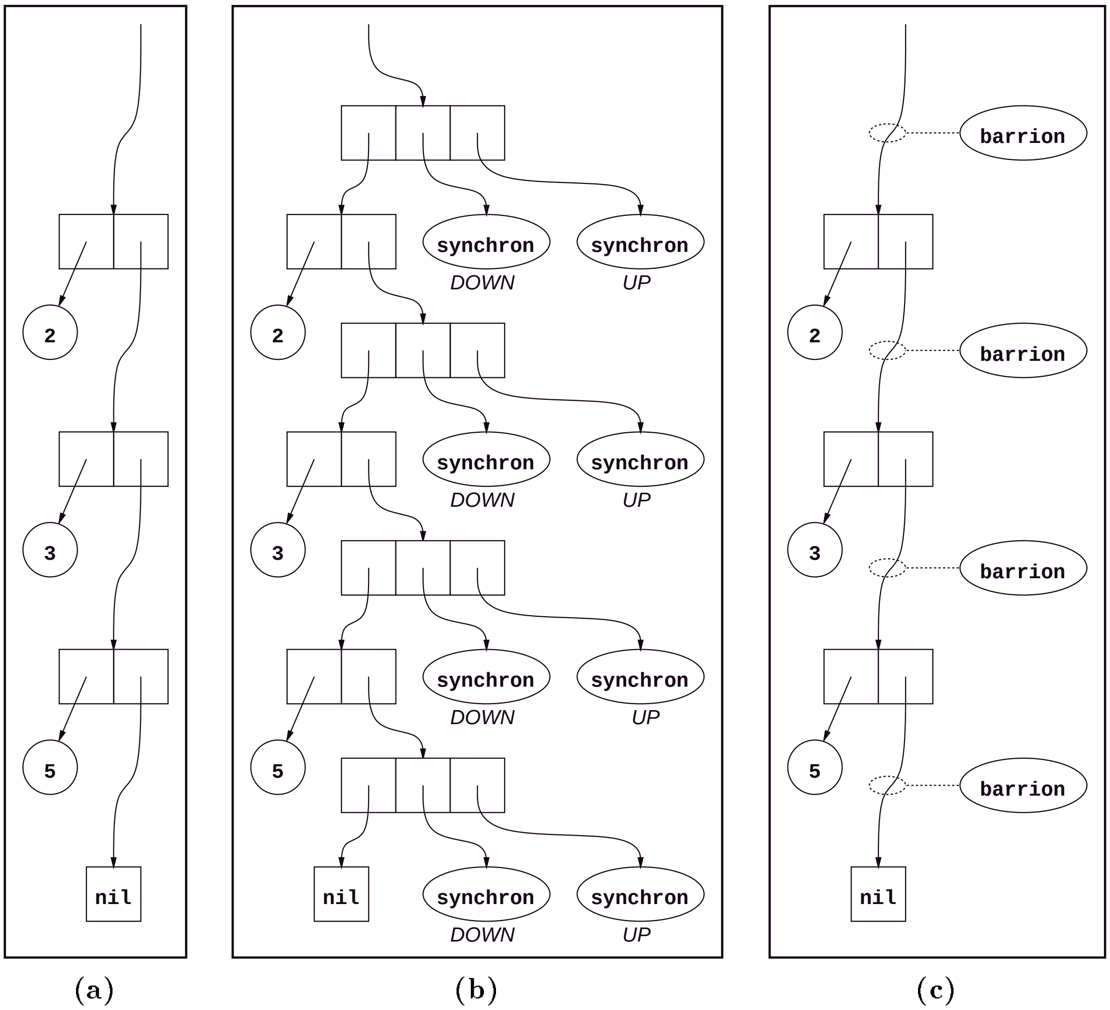


Figure 5.12: Comparison of list structure and synquence structure. **(a)** is a list of three elements. **(b)** is a synquence of three elements. **(c)** is an abbreviation of synquence **(b)**.

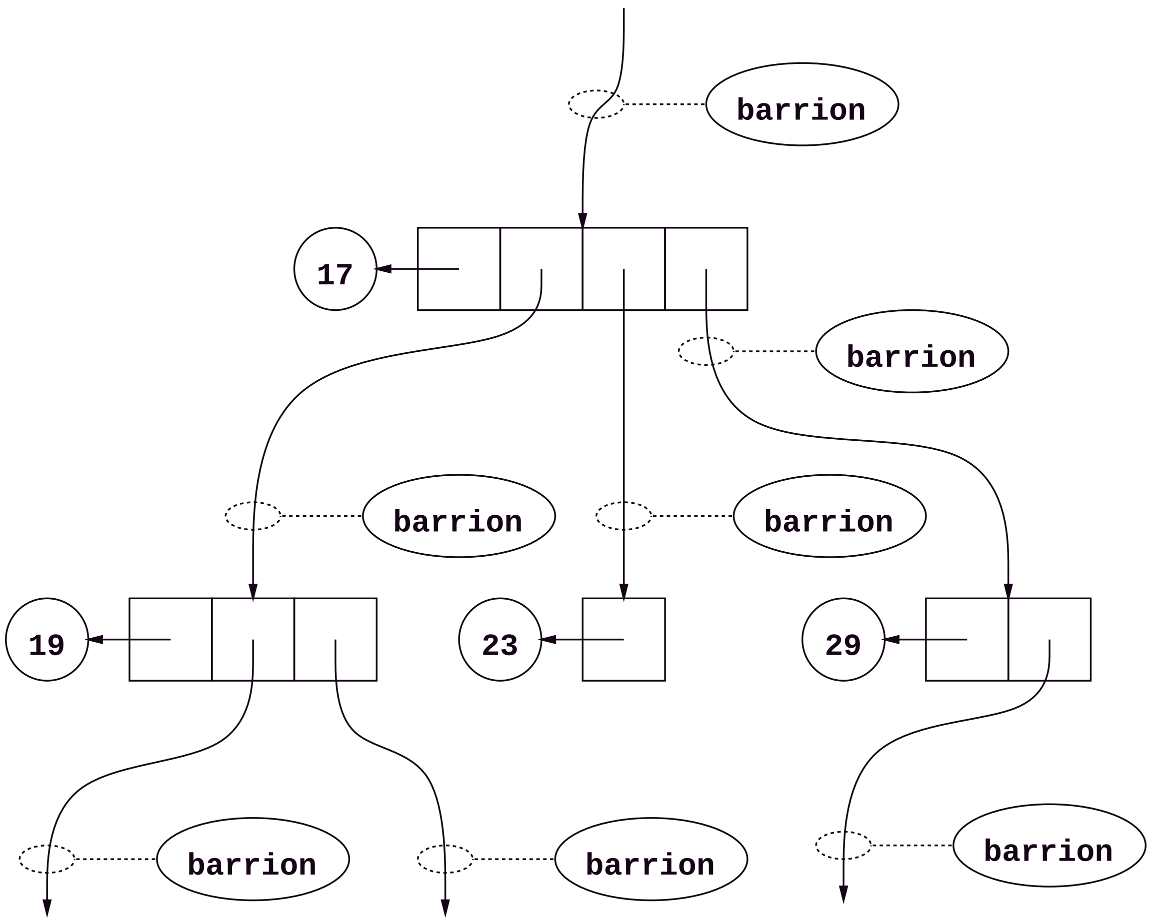
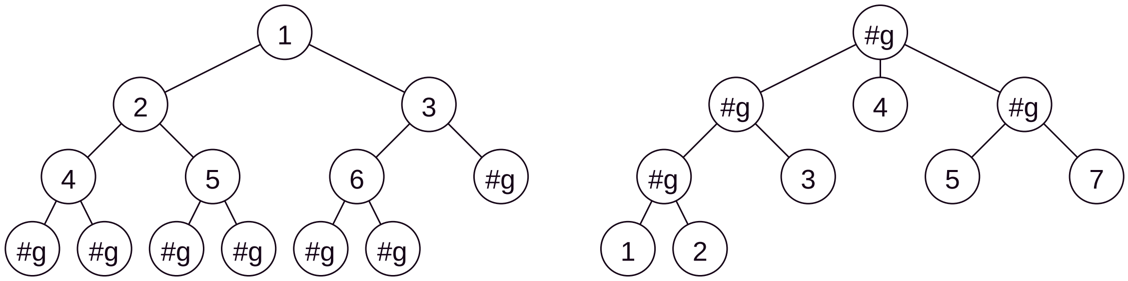


Figure 5.13: Structure of a sample syndrite.

less node, but a childless node. Trees with null leaves (like the one pictured in Figure 4.20) are represented as syndrites whose leaves carry null or gap values (Figure 5.14(**a**)). The common case of trees with valued leaves but valueless internal nodes is handled by syndrites with gaps at the non-leaf nodes (Figure 5.14(**b**)).



(**a**) A tree with valueless leaves. (**b**) A tree with valueless internal nodes.

Figure 5.14: Using gaps (`#g`) to represent valueless nodes.

5.3.3 Synchrons

A synchron is an entity that represents the down or up barrier of a call boundary. It is the fundamental mechanism by which slivers synchronize with each other to achieve lock step processing. We assume that synchrons are propagated in such a way that all slivers in the same lock step component share access to the same synchron for the same call or return event. For example, Figure 5.15 indicates the sharing of synchrons (represented by barrions) in a simple sliver network. We shall see shortly how this sharing is achieved. Right now we concentrate on what must be true of the synchrons themselves.

Synchrons support the following operations:

- *Create* makes a new synchron from scratch. This operation is invoked by generating slivers when constructing a slag.

- *Unify* glues two synchrons together so that they become the same synchron. Slivers with fan-in invoke this operation on the corresponding synchrons of their inputs. This

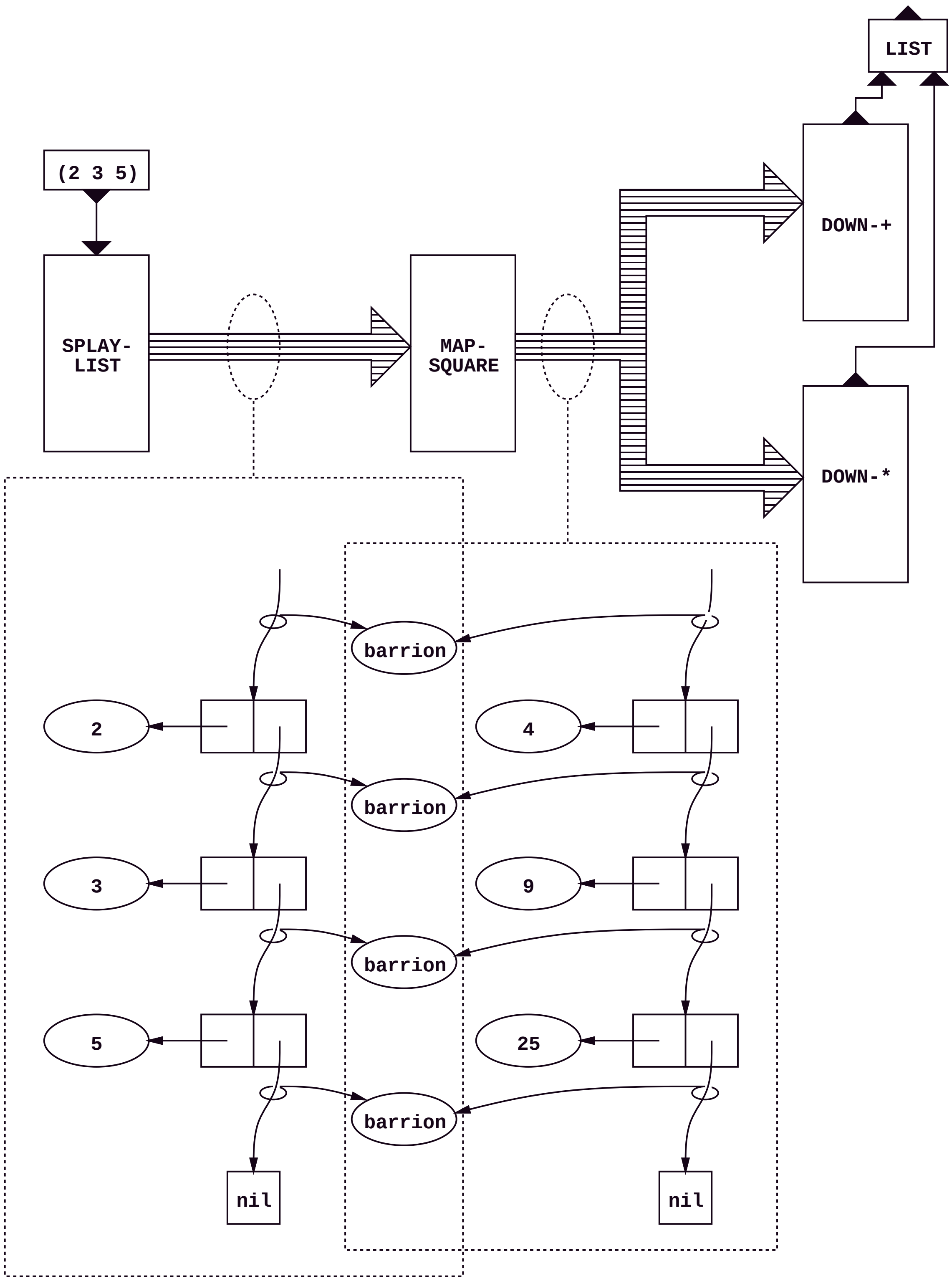


Figure 5.15: Barrion sharing in a simple linear computation. Two intermediate synquences share the same synchrons, permitting all four slivers to synchronize with each other.

guarantees that synchrons generated independently in different parts of the same lock step component will eventually come to denote the same barrier.

- *Wait* is called by a sliver on a synchron when it is ready to participate in a *rendezvous* with all the other slivers in the network. The rendezvous occurs when all the slivers in a lock step component have called *wait* on a shared synchron. It is only possible to rendezvous once at a given synchron. The details of how a rendezvous is determined are rather subtle; they are expanded on in Chapter 7 and formally described in Chapter 8.

- *Precede* declares that the rendezvous at one synchron must occur before the rendezvous at another synchron. It turns out that this operation is necessary to ensure that up synchrons rendezvous in the proper order in the presence of filtering.

5.3.4 Slag Dynamics

The slag diagrams in Figures 5.12, 5.13, and 5.15 are somewhat misleading because they don't accurately portray the time-dependent nature of slags. For example, in Figure 5.15, neither synquence ever exists as a complete entity at any point in time. Instead, a synquence grows downward only as new elements are demanded; meanwhile, it shrinks from above because slivers eagerly drop references to skeletal nodes as soon as they can. In fact, at most one skeletal node of a synquence actually exists at any point in time![9]

Figure 5.16 presents a selected sequence of snapshots that illustrate how the synquences from Figure 5.15 actually unfold over time. The dotted boxes containing question marks represent suspended synquence computations that do not resume until after a rendezvous has taken place on the down synchron of the barrion. The snapshots indicate how the lazy nature of slags leads to a constant storage requirement for this example.

Suppose we modify the two accumulators from the previous example to have up shape rather than down shape. Figure 5.17 shows a snapshot of the modified computation when it first reaches the end of the list. Even though the intermediate synquences have disappeared,

[9] Clarification: By "exists" I mean "is accessible". I assume throughout that inaccessible structures that will be reclaimed by the garbage collector do not count towards the space consumed by a computation.

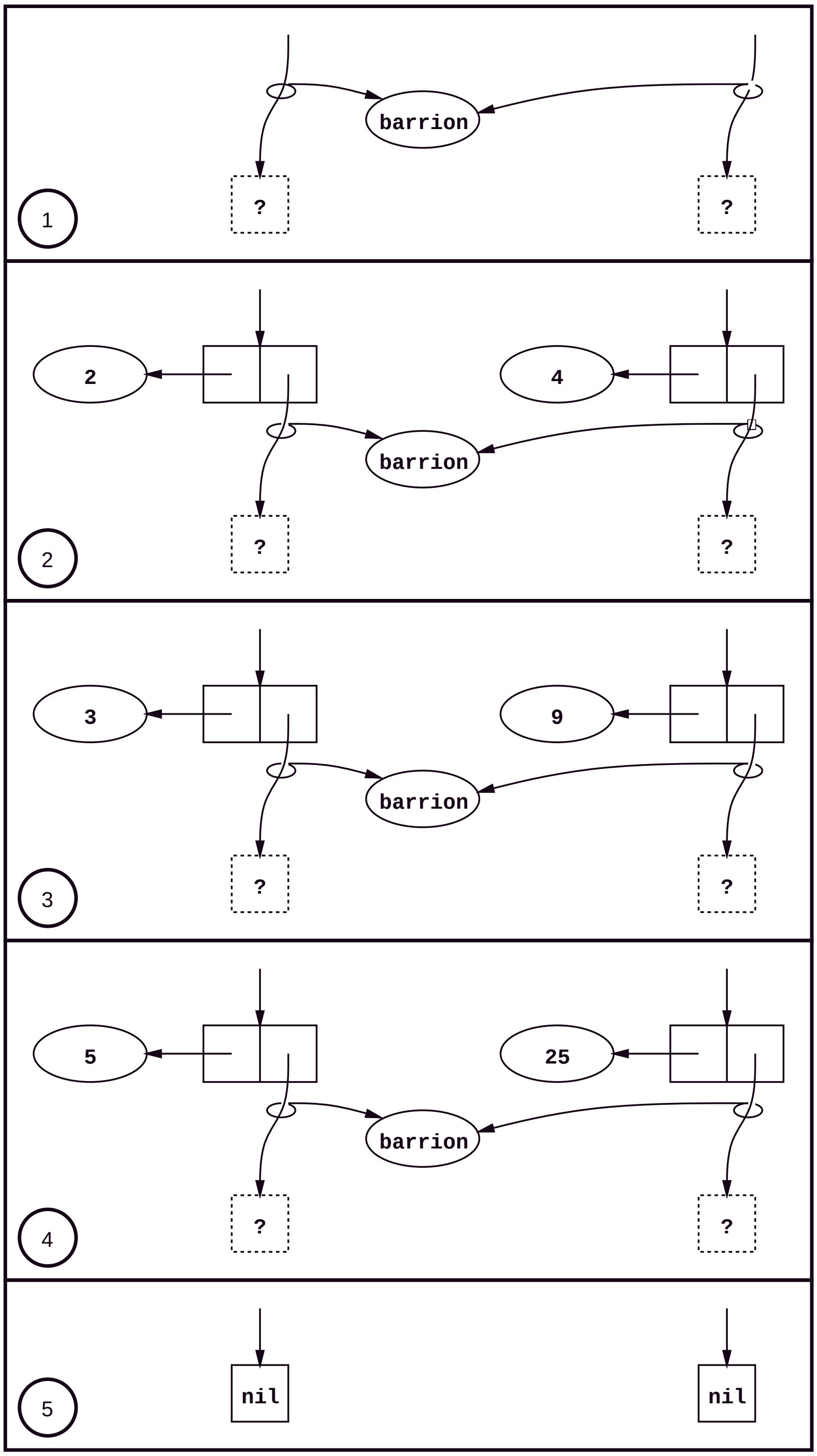


Figure 5.16: A “movie” showing how synquences unfold over time.

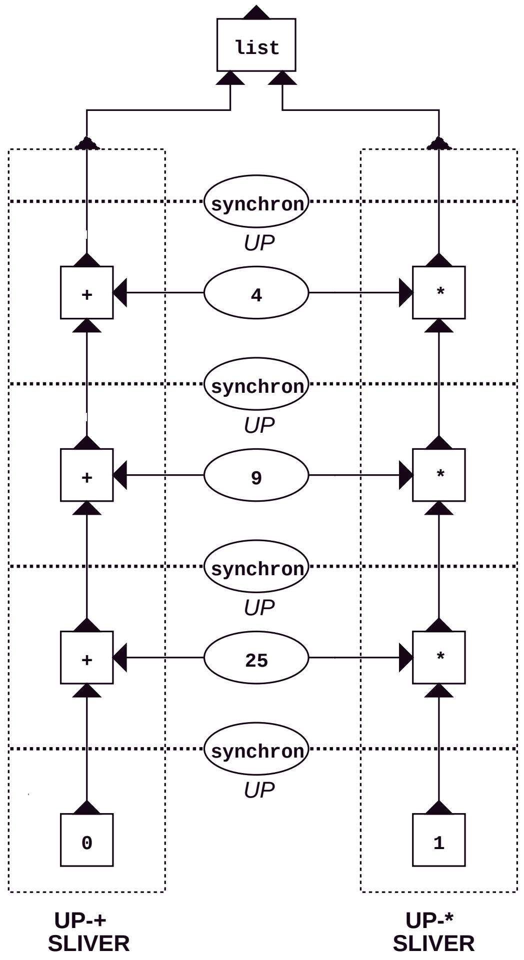


Figure 5.17: A snapshot illustrating how parts of a synquence can be stacked by slivers.

each of the accumulating slivers maintains its own stack of values and up synchrons. This means that the modified computation requires space linear in the length of the list argument. The up synchrons shared by the stack couple the remaining `+` and `*` operations so that they will be performed in lock step as the stacks are popped.

Computations involving syndrites are trickier to analyze. In computations that correspond to sequential tiles, a syndrite at any point in time is unfolded only along the branch currently being explored; branches previously explored have disappeared and those not yet explored are suspended. For computations that correspond to parallel tiles, a syndrite may in the worst case be completely unfolded. Although undesirable, this accurately models the space consumption of a monolithic tree-parallel computation. The storage requirements in this case can be greatly reduced by sequentializing the parallel computation.

5.4 Slivers Revisited

5.4.1 Sliver Classification

Slivers are just slag-manipulation procedures that observe some important local structural and behavioral constraints (described later). The slags consumed and produced by a sliver correspond a subtile's consumpts and products; the non-slag inputs and outputs of a sliver correspond to a subtiles arguments and results. To maintain these distinctions, we will say that a sliver *consumes* input slags, *produces* output slags, *takes* arguments, and *returns* results.

Slivers are classified into three categories according to how they manipulate slags:

1. *Generators.* A generator is a sliver that consumes no slags but produces one or more slags. Sample generators include a sliver that produces a synquence of integers between two limits and a sliver that converts a tree into a syndrite. Generators are responsible for creating fresh synchrons for the slags that they produce.

2. *Transducers.* A transducer is a sliver that both consumes and produces slags. A transducer with multiple slag inputs must unify the corresponding synchrons of its inputs and use the unified synchrons in its outputs. Common transducers include:

- *Mappers* that apply a function elementwise to a slag.
- *Filters* that selectively replace slag elements by gaps. (We will see below that, due to technicalities, filters cannot simply be represented as mappers.)
- *Scanners* that produce output slags containing the intermediate values of an accumulation over inputs slags.
- *Truncaters* that produce potentially truncated versions of their input slags.

3. *Reducers.* A reducer is a sliver that consumes input slags but returns results instead of producing output slags. Reducers include *accumulators* that accumulate a value from a slag and *selectors* that select an element from a slag. For instance: a sliver that sums the elements of a synquence, a sliver that collects syndrite elements into a tree, and a sliver that returns the last element of a synquence.

5.4.2 Sliver Requirements

Not every procedure that manipulates slags has what it takes to be a sliver. In order to be a sliver, a slag-manipulation procedure must observe some important local structural and behavioral requirements. Here we describe the requirements and discuss the behavior that that they engender.

A sliver can be viewed as a fragment of a monolithic recursive procedure in which every access to the next slag (via a synter) coincides with a recursive procedure call. We will associate each recursive call in a sliver with the down and up synchrons annotating the current consumpt and product slags.

Every sliver must obey the following requirements:

1. *Synchron propagation.* A sliver must ensure that all of its output slags use the same synchrons (in the same order) as any of its input slags (as in Figure 5.15). This guarantees that all slivers in a lock step component share the same synchrons. If a sliver has no input slags, it generates fresh synchrons that are then shared by its output slags. If a sliver has multiple input slags (which may have been produced by independent generators) it uses the synchron *unify* operation to combine the input synchrons into a single output synchron.

2. *Down synchronization.* As a part of making a local recursive call, a sliver must *wait* for a rendezvous at the down synchron with all the other slivers in the lock step component. This rendezvous simulates the down barrier of a monolithic recursive call. After the rendezvous, the slivers compute independently (modulo data dependencies) until the next rendezvous.

3. *Up synchronization.* If a sliver returns from a local recursive call, then it must *wait* for a rendezvous at the up synchron with all the other slivers that return from their calls. This rendezvous simulates the up barrier of a monolithic recursive call. However, a sliver that makes a tail-recursive call never returns, so it will *not* rendezvous with the other slivers. Thus, a lock step component of linear slivers only exhibits stacking behavior when at least one of its slivers has *up* shape. When all the slivers have *down* shape, the entire lock step component acts as an iterative computation. In this important case, slivers behave like the series procedures in Waters's series package. But slivers also preserve the shape aspects of tree computations as well. For example, a sequential tree computation built only out of slivers with *pre* shape has the tail-recursive characteristics of a pre-order accumulation (see Figure 4.24 on page 4.24).

4. *Lazy elements.* The demand-driven nature of the underlying computational model implies that an element wire does not transmit a value unless it is requested. In other words, while the call boundaries of subtiles are strict, the side-to-side product/consumpt boundaries are non-strict. Operational faithfulness dictates that slivers must delay the computation of every slag element so that the element value is never computed if it is never requested.

5. *Aggressive reference dropping.* In order to preserve expected storage behavior, slivers must aggressively drop references to slags, as well as to the values and synchrons held by slags, when they can no longer be referenced. For example, if a sliver only requires one component of a slag, it must extract that component as soon as possible so that the other components become inaccessible (and therefore garbage-collectible). Aggressive extraction is required not only for preserving desirable space requirements, but also for avoiding spurious deadlock (see the deadlock discussion in Chapter 7).

6. *Appropriate treatment of gaps.* Due to the presence of filters, a sliver may sometimes discover a *gap* — a token indicating that the element at the given position has been filtered out. The sliver must handle the gap in a reasonable way. The conventions for handling gaps will be discussed in Section 5.5.2.

In practice, it is challenging to write procedures that embody all of the sliver requirements listed above. The sources of difficulty will be discussed in Chapter 7. However, it is possible to express a wide range of common sliver patterns as instances of a handful of carefully written sliver templates. The SYNAPSE language described in Chapter 6 is an example of this approach.

Note that the sliver requirements do not include any rule that corresponds to the unitilable assumption invoked in the discussion of tiles and subtiles. This assumption was made purely to simplify the presentation. As long as slivers obey the above requirements, nothing prevents them from being implemented in terms of nested loops or mutual recursions.

A rendezvous between slivers resembles interprocess synchronization in many models of concurrent processes (e.g., [Hoa85, Mil89, CM90]). However, there are several aspects that distinguish sliver synchronization from these other models.

- With slivers, communication and synchronization are decoupled. Communication is achieved by referencing a data structure, while synchronization is achieved by applying *wait* to a synchron. This approach contrasts with models in which every communication event synchronizes sender and receiver.

- Slivers engage in a multiway rendezvous that involves all the slivers in a lock step component. Most synchronous communication models support only a two-way rendezvous. While CSP [Hoa85] supports a multiway rendezvous, it is limited to communication between a single sender and multiple receivers.

- Since slags carry both down and up synchrons, slivers can naturally express computations (including tree-shaped ones) that can rendezvous upon return from a call return as well as initiaion of the call. This contrasts with the linear iterative nature of many concurrent process models.

5.4.3 Sliver Dynamics

The synchronization requirements obeyed by slivers constrains them to work together in a way that mimics the behavior of a monolithic procedure. As a simple example, consider an iterative list-averaging program expresssed as a sliver network:

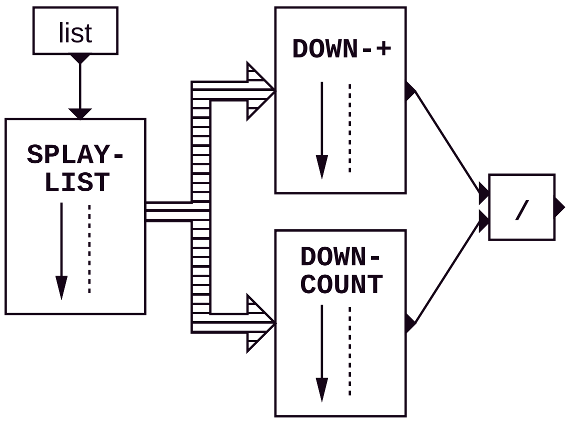


Figure 5.18: Sliver diagram for a program that iteratively computes the average of a list.

`SPLAY-LIST` converts a list to a synquence; `DOWN-+` iteratively sums a synquence of numbers; and `DOWN-COUNT` finds the length of a synquence. Each of the slivers maintains one state variable of the iteration (current list, current sum, and current count). The diagram exhibits fan-out because the synquence output of `SPLAY-LIST` is use as an input to both `DOWN-+` and `DOWN-COUNT`.

A "movie" of selected computation snapshots for the network on the input list `(7 2 6)` appears in Figure 5.19. As indicate by the snapshot movie, `average-list` is indistinguishable in behavior from a constant-space iteration in three state variables. This is remarkable because, as explained in Chapter 3, classical lazy data techniques would build up space linear in the size of the input for this example. In contrast, synquence version is *guaranteed* to use constant space due to the way that slivers make use of the synchronization information supplied by synchronized lazy aggregates.

Indeed, although slivers are composed in the aggregate data style, they behave more like concurrent processes communicating via unbuffered channels. However, as discussed in

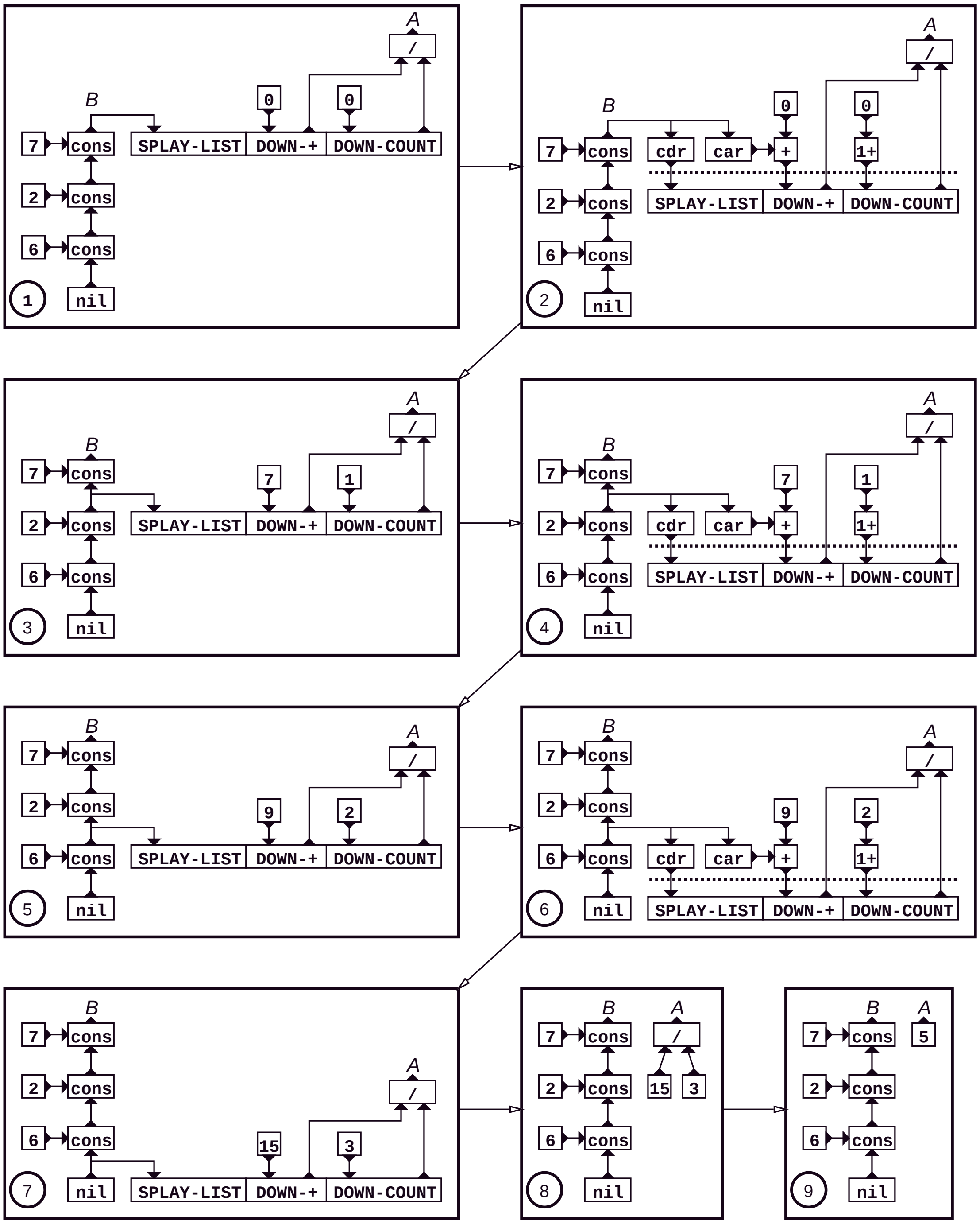


Figure 5.19: A stylized "movie" of snapshots from the sliver-based *down* computation of the average of the list `(7 2 6)`. Assume that A has requested the result. The dotted horizontal lines represent a down synchron shared by the slivers. The list is represented as the result of `cons` operators (see Chapter 8). The internal details and connectivity of the three slivers have been suppressed. Garbage collection of the input list structure is not shown because in general it might be accessible from some other point B in the program.

the previous chapter, the channel-based approach would require some sort of copy process to fan the result of the generator out to the two accumulators. Slags make it is possible to express sliver networks via the same mechanisms used to express other data dependencies in a given language.

We can modify the two *down* accumulators in the list-averaging problem to be *up* accumulators. Then the resulting network behaves like a recursive procedure of one argument (current list) that returns two results: the sum of the numbers in the current list, and the length of the current list. Figure 5.20 shows a snapshot movie for the modified computation. The movie indicates how the computation `cdr`s down the list in *down* phase, and then processes the two results in lock step in its *up* phase.

Of course, we could also modify the list-averaging network so that only one of the *down* accumulators were changed to an *up* one. The resulting computation would have an *up* shape, but would exhibit characteristics of both of the previous examples.

Tree computations and more interesting linear computations have too high a "visual complexity" to illustrate their dynamics in the movie format. We will study such computations through a text-based interface in Chapter 6. I have also developed a dynamic program animator (the DYNAMATOR) that automatically animates computations using a visual representation similar to that in Figures 5.19 and 5.20. The animator is described in Chapter 9.

5.5 Filtering

The model of sliver decomposition presented so far glosses over a few important complications. All of these complications are related to filtering, which is surprisingly difficult to handle in both an operationally faithful and reusable manner. Here we discuss filtering in more detail and extend the sliver decomposition technique to handle the complications that it introduces.

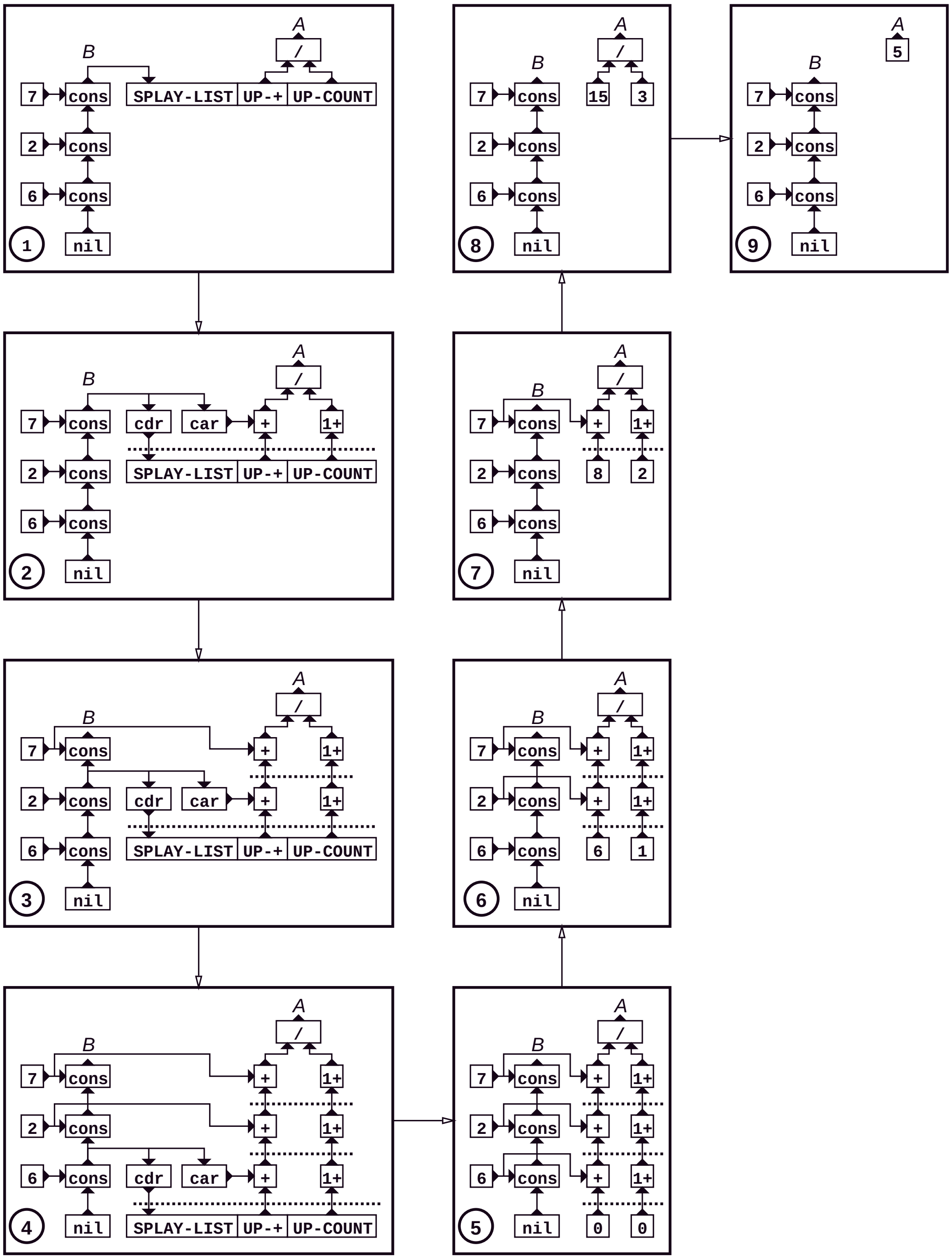


Figure 5.20: A snapshot movie of an *up* computation of the average of the list `(7 2 6)`. The long horizontal dotted lines represent shared down synchrons. The short horizontal dotted lines represent shared up synchrons. Note how the up synchrons force the otherwise independent stacks of pending operations to pop in lock step.

5.5.1 Gaps

A filter is a transducer that *passes* to the output slag only those elements from the input that satisfy a given predicate. Elements that do not satisfy the predicate are said to be *blocked*.

There are two basic approaches to filtering:

1. In the *compaction approach*, the output of a filter contains only the passed elements. For example, a compacting version of a filter that passes odd numbers would transduce the input list `(8 3 2 4 7)` into the shorter output list `(3 7)`. The compacting approach to filtering is commonly employed in aggregate data techniques for linear structures and in channel-based techniques.

2. In the *gap approach*, the output of a filter contains not only the passed elements, but also "holes" indicating where the blocked elements used to be. A gap version of an odd number filter would transduce the input list `(8 3 2 4 7)` into the output list `(#g 3 #g #g 7)`, where `#g` is a distinguished entity that represents the lack of a value. Gap filtering is used in situations where the location of elements is important. For example, languages with array-based or data-parallel features (e.g., [Ive87, Ame89, GJSO92, Sab88, Ble90, RS87, Ble92]) present models in which elements reside at a particular location in an data structure. Compaction filtering is typically not the default in these approaches because (1) elements are often indexed by their location and (2) moving an element between locations may be expensive (e.g., it may require interprocessor communication). Instead, blocked locations either hold a distinguished value, or they are tagged so that they do not participate in calculations.

The compaction approach is straightforward for linear structures, but does not naturally extend to more complex structures like trees and multi-dimensional arrays. Using compaction to filter out an internal node of a tree or two-dimensional array requires the other elements to be rearranged in arbitrary and non-trivial ways. In contrast, the gap approach works for structures with arbitrary topology since the filter always preserves the connectivity of the input.

Sliver decomposition employs the gap approach to filtering primarily because of the lock step nature of sliver processing. A synquence element not only appears at a particular index in the corresponding a sequence (or tree) of values, but it also appears at a location in the synquence that is conceptually bounded by a pair of barrions. Since all slivers in a lock step component share the same barrions, every location of an input synquence to a filter must be preserved in its output. In particular, even the locations of elements blocked by a filter must appear in the output. The location-dependent nature of synquences makes the gap approach the natural choice. In addition, gapped filtering easily generalizes to the tree structure of syndrites.

A SYNAPSE filter inserts a distinguished *gap* value (written `#g`) at every location corresponding to a blocked input element (see Figure 5.21). In terms of the subtile model introduced in Section 5.2.1, a gap corresponds to a triple of wires in which the termination wire and presence wire are both false but the element wire is unconstrained (i.e., its value doesn't matter).

5.5.2 Gap Conventions

Slivers must be carefully designed to handle gaps in an appropriate manner. Since gaps stand in for elements that have been blocked by a filter, a sliver should treat them as if they aren't there. This suggests that the functional arguments of a sliver should not be applied to gap values, and leads to two rules of thumb for handling gaps:

1. *Gap contagion*: Transducers should treat gaps in a contagious manner. Mappers, filters, truncaters, and shifters should map every location with an input gap to an output gap. For transducers that take more than one input slag, an output location should hold a gap if any of the corresponding input locations hold a gap.

2. *Gap apathy*: Linear and sequential tree reducers should ignore all gap positions when accumulating or selecting a result. For example, a length accumulator for the gapped list `(#g 3 #g #g 7)` should return `2`, not `5`. The second element of this list is `7`, not `3`. For reducers that take more than one input slag, all input location should be ignored if any of the corresponding input locations hold a gap.

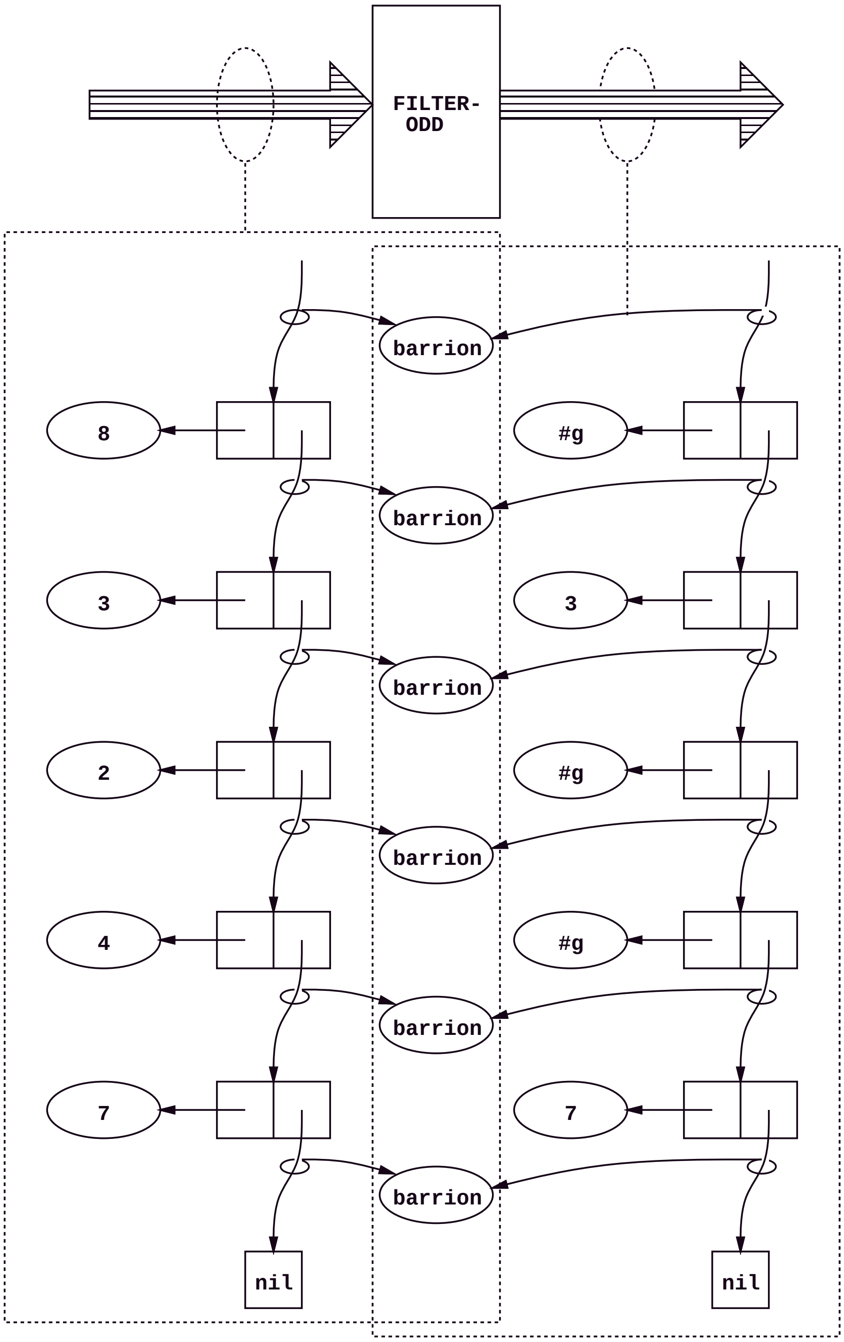


Figure 5.21: Distinguished gap values fill the locations in a synquence that correspond to elements blocked by a filter.

Unfortunately, these heuristics don't handle some important situations:

- Scanners are an important exception to the gap contagion principle for tranducers. Recall that a scanner annotates every node of a slag with the current value of an accumulator when an accumulator processes the node. Since the accumulator has a value even when a gapped node is reached, it is meaningful to annotate gapped nodes as well. In fact, several examples in Chapter 6 require scanners to provide this capability. By default, then, scanners produce slags that contain no gaps. If desired, it is always possible to reintroduce gaps by applying a two-input mapper to the input and output of the scanner (see Figure 5.22).

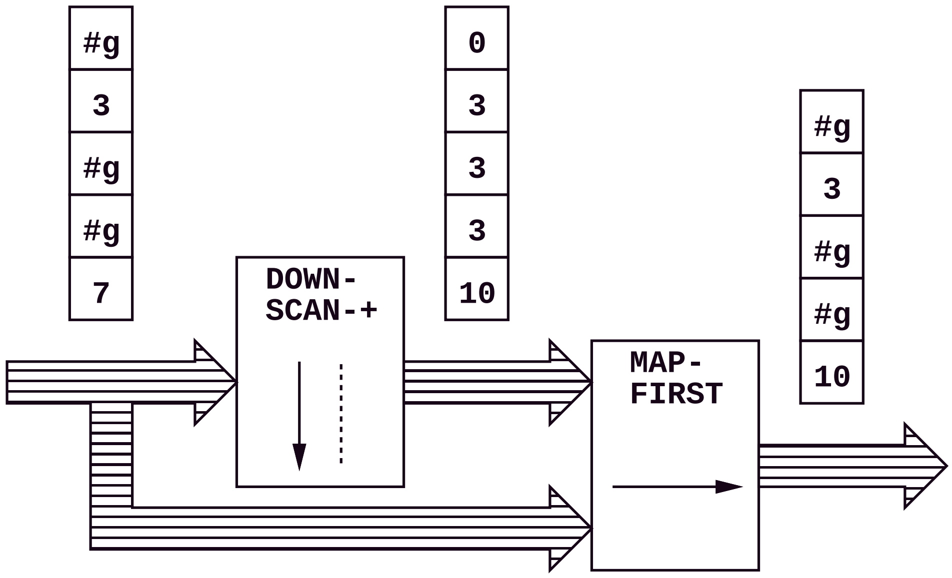


Figure 5.22: Scanners do not obey the gap contagion principle. However, the contagion property of two-input mappers can be used to reintroduce gaps into the result of a scan. Here, `MAP-FIRST` is a mapper that returns the elements from its top input.

- In certain situations it is desirable for the functional arguments to slivers to be able to manipulate gaps with the same status as other elements. For example, consider a numerical slag A that is filtered into three slags B, C, and D that contain, respectively, its positive, zero, and negative elements. Suppose we want to design an accumulator that takes B, C, and D as inputs and performs an operation at each location that depends on which slag has a non-gap element at that location. Since two of the inputs will have a gap at any given location, the gap apathy rule implies that no operation can be performed at any location! This is clearly absurd.

We will employ two methods for addressing this problem:

1. One approach is to provide mechanisms for *reifying* otherwise implicit gaps into explicitly manipulable gaps and *unreify* the explict gaps back into implicit ones. When necessary, higher order sliver functions can be designed to test for reified gaps. For instance, in the positive/negative/zero example, the accumulator could be applied to gap-reified versions of B, C, and D, and it could test for the presence of explicit gaps in order to determined which operations to perform.

2. An alternative is to design classes of slivers that handle gaps in special ways. For example, given a list of slags, a commonly desired processing pattern is to (1) check the slags in turn until one is found that has a (non-gap) element at the current position and (2) perform an action that may depend on which slag provided the element. This pattern could be captured by a specially designed sliver that ignores the gap apathy heuristic. Such a sliver would be able to handle the positive/negative/zero example described above.

The reify/unreify approach is general, but leads to sliver programs that have a brute force character. The special-purpose sliver approach leads to more elegant sliver programs, but is more ad hoc.

- Parallel tree reducers are trickier to handle than their sequential counterparts. Consider the behavior of a *binary up* accumulator at the node of a binary syndrite. Suppose that that the accumulated values from the left and right branches are l and r, respectively. If the node holds a non-gap value v, then the accumulator needs to combine three values: v, l, and r. But if the node holds a gap, there are only two values to combine: l and r. Since the same combiner cannot work for both cases, it is necessary to specify *two* combiners for a parallel tree accumulator.

How to handle selection for a gapped syndrite is less clear. In a gapped synquence, an element is specified by a single index, and gaps can simply be ignored in finding the (ungapped) element at the give index. However, a syndrite element is generally specified by a path that describes what branches to make at each node of the tree. If a node on the path holds a gap, the node cannot readily ignored because it generally will have more than one branch; which should be followed? If we just use the path information to choose a branch at every gapped node, then the path may ultimately terminate at a gapped node. In this case, what is the value of the selector supposed to be? Because of these difficulties, we provide no parallel tree selectors.

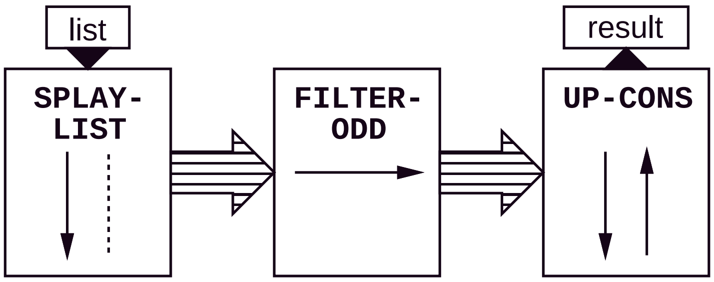
5.5.3 Reusability

Why aren't filters just a particular kind of map? The above discussion on gap conventions provides a partial reason: since gaps are "holes" in a slag, it's desirable for mapping functions never to deal with gaps at all. In particular, they shouldn't take them as arguments or return them as results; this eliminates expressing filter as a kind of map. But this explanation is somewhat unsatisfying because the gap reifying/unreifying mechanism introduced above could be invoked for the special case of filters.

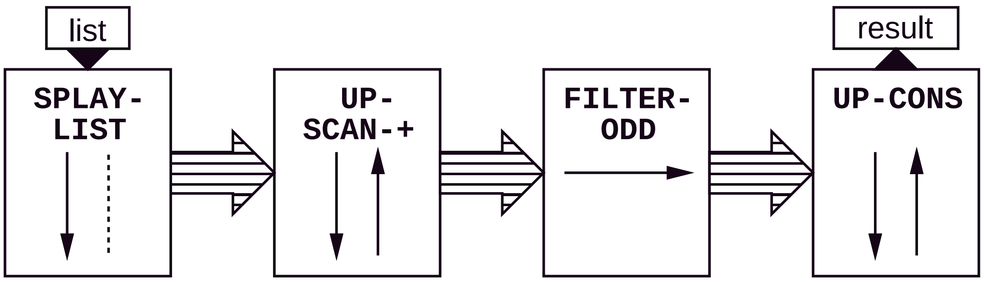
The real reason that filters aren't just maps is that the reusability goal for slivers requires filtering to be a more involved process than suggested by the simple gap model presented above. Using this model, it is impossible to design transducers and reducers that exhibit expected space behavior in every shape-compatible context. We demonstrate this fact via some concrete examples.

Consider the two sliver diagrams shown in Figure 5.23. In both diagrams, an `UP-CONS` sliver consumes the synquence produced by the `FILTER-ODD` sliver. Here is the behavior we expect for the diagrams:

1. The computation described by diagram **(a)** should return a list of the odd elements in the input list. It should test all of the list elements for oddness in the *down* phase. At the completion of the *down* phase, the space required by the computation should be proportional to the number of *odd elements* in the input list. In particular, if



(a)



(b)

Figure 5.23: Two sliver diagrams in which `UP-CONS` follows `FILTER-ODD`. As explained in the text, it is hard to design `FILTER-ODD` and `UP-CONS` so that both diagrams give rise to the expected computations.

the input list contains only even numbers, the *up* phase should require no space for pending computations, since there shouldn't be any.

2. The computation described by diagram **(b)** should return a list of the odd numbers resulting from a upward sum scan of the input list. Since the inputs to the filter are computed in the *up* phase, all of the tests for oddness must be performed in the *up* phase. At the completion of the *down* phase, there must be a pending `if` for every element produced by `up-scan-+`. The computation always requires space linear in the length of the list at this point, even if the input list contains only even numbers.

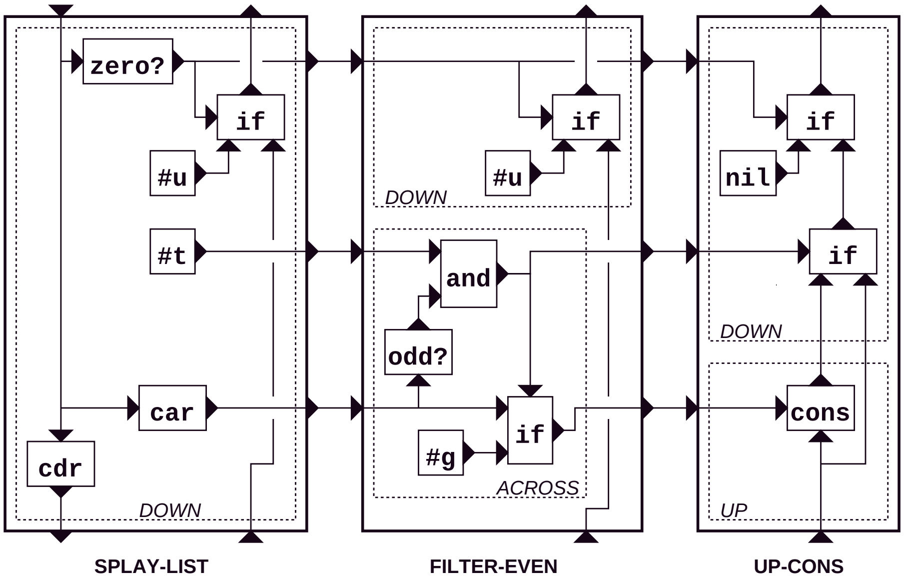
Using the standard sliver implementations developed earlier in this chapter for these two examples leads to the subtile diagrams in Figure 5.24. Unfortunately, the shaded path in subtile diagram **(b)** indicates the presence of deadlock. The problem is that the lower `if` in `UP-CONS` is a *down* operation that must perform its test before the subcall, but the test depends on the value of the result of the subcall of `UP-SCAN-+`.

It is possible to fix the deadlock in example **(b)** by modifying `UP-CONS` to perform the test of the lower `if` *after* the subcall has returned rather than before it. Figure 5.25 depicts the `UP-CONS-TEST-UP` sliver that results from this modification. `UP-CONS-TEST-UP` uses a `seqn` node to force the subcall to precede the test performed by the lower `if`.[10] Replacing `UP-CONS` with `UP-CONS-TEST-UP` in Figure 5.24**(b)** yields a specification for a computation with the desired behavior.

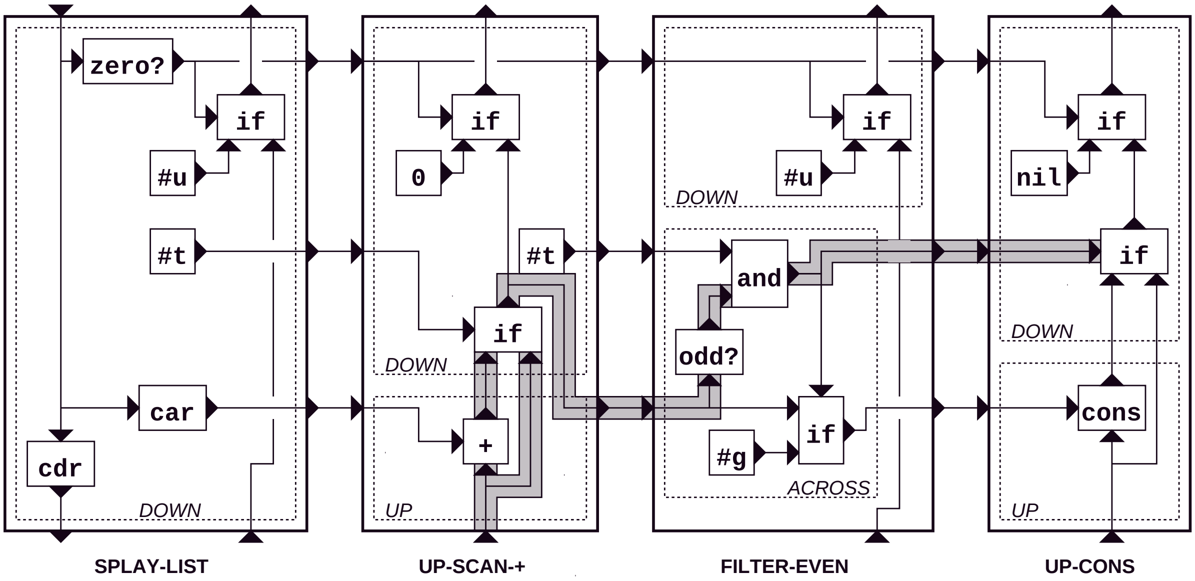
Unfortunately, if `UP-CONS-TEST-UP` replaces `UP-CONS` in subtile diagram **(a)**, the wrong behavior results. `UP-CONS-TEST-UP` guarantees that oddness tests are only performed in the *up* phase of the computation. This means that the resulting computation always requires space linear in the length of the input list at the completion of the *down* phase, even if the list contains only even numbers. This is inconsistent with the expected behavior of example **(a)**.

This example illustrates a tension between operational faithfulness and reusability. Operational faithfulness requires us to have accumulators like `UP-CONS` that preserve the tail-

[10] Recall that a requested `seqn` node first requests the value of its left input wire, and only when that wire has returned a value does it reroute its request to the right input wire. In this example, the request on the left subwire forces the subcall to occur before a request is propagated to the `if`.



(a)



(b)

Figure 5.24: The details of the two filtering problems from Figure 5.23. The shaded path in **(b)** indicates the presence of deadlock.

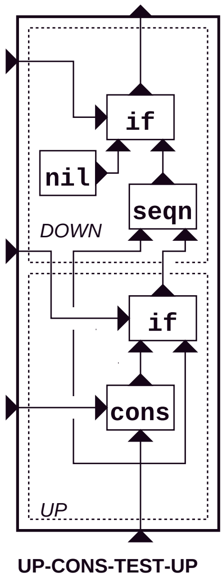


Figure 5.25: `UP-CONS-TEST-UP` results from the addition of a `seqn` operation to the `UP-CONS`. The `seqn` changes the lower `if` from a *down* operation to an *up* operation by forcing the subcall to return before the `if` is tested.

recursive nature of filtering and accumulators like `UP-CONS-TEST-UP` that can handle filters performed in the *up* phase. But reusability suggests that we should have only one *up* accumulator for `cons`, not two. This leaves us with a choice:

1. *The two-up approach*: Accept two versions of every *up* accumulator: one which performs gap tests in the *down* phase, and one of which performs gap tests in the *up* phase.

2. *The one-up approach*: Redesign filtering so that a single *up* accumulator suffices in both situations.

The two-up approach might seem simpler, but it leads to a disaster. The problem is that the prospects for reusability in the presence of filtered *up* scanners are even grimmer than indicated by this particular example. The result of a filtered *up* scan is essentially a different "type" of slag than the one used in all the other subtile decompositions. For every *across* or *up* sliver S that takes an input of the original slag type, the two-up approach requires developing a new sliver S' that takes an input of the new type. The situation is even worse for multi-input slivers: if a sliver takes n slag inputs, it is necessary to have 2^n different versions of the sliver to account for all combinations of argument types. (Usual polymorphism mechanisms are of no help here because each of the potential combinations actually works in a different way.) Finally, even this brute force fails to accurately model the expected behavior of computations like the one pictured in Figure 5.26. The problem in this case is that an ideal `UP-CONS` should only perform gap testing in the *up* phase for those slag locations at which the result of `SPLAY-LIST` had odd elements. Neither `UP-CONS` nor `UP-CONS-TEST-UP` is sophisticated enough to exhibit this behavior.

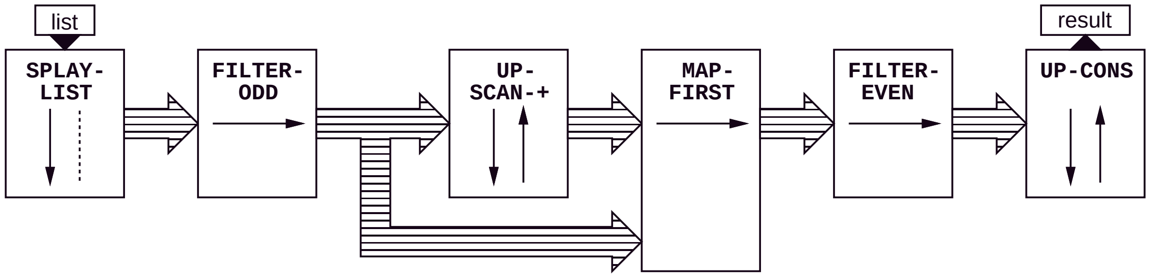


Figure 5.26: Sliver diagram whose expected behavior is difficult to model in the two-up approach.

The numerous problems with the two-up approach suggest that the one-up approach is a better tack. In Section 7.2.4, I describe a design for a one-up approach that works for synquences. The details of a one-up approach for syndrites have not yet been figured out.

5.5.4 Up Synchronization

Preserving operational faithfulness for the *up* phase of a computation in the presence of filtering turns out to be rather challenging. We would like to achieve the following two goals:

1. *Operation order*: The up synchrons of a synquence or syndrite have a natural time ordering from bottom up. Operations should respect this time ordering.

2. *Tail calls*: If all the slivers in a lock step component locally make tail calls at corresponding call boundaries, the entire lock step component should behave as if a

monolithic tail call were made. In particular, no pending operations or stack space should be associated with such a call.

Unfortunately, filtering makes it difficult to satisfy both of these goals. Figure 5.27 illustrates various approaches to a computation with two *up* slivers accumulating `f` and `g`. Due to filtering performed in the *down* phase, there are many locations at which accumulations do not have to be performed.[11]

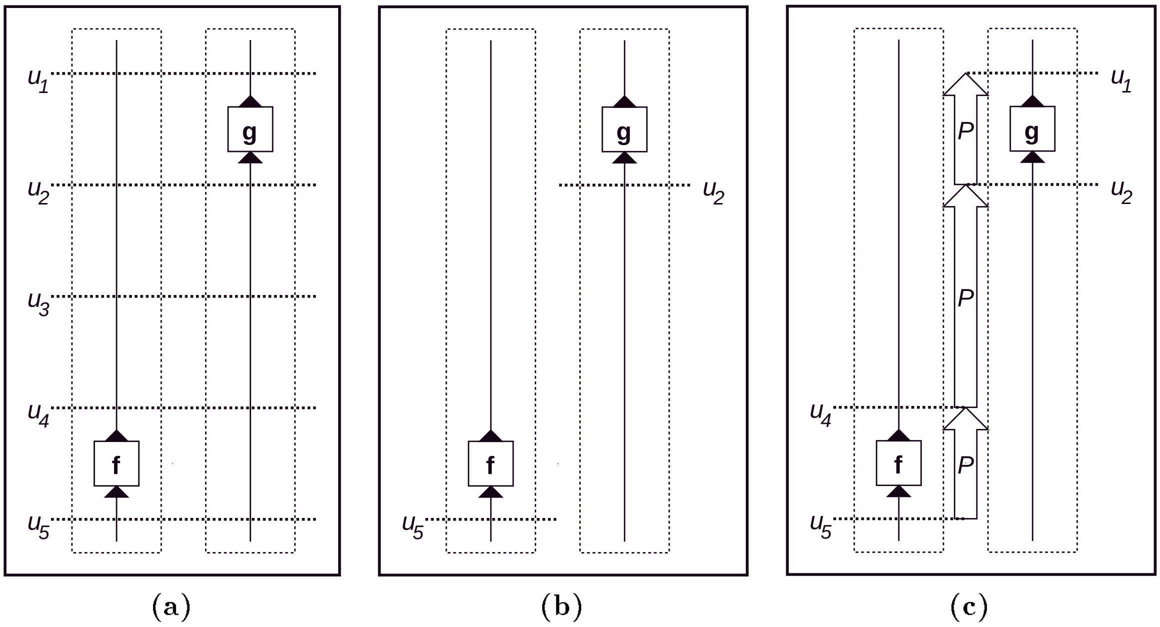


Figure 5.27: Different approaches to handling up synchronization in the presence of filtering.

Figure 5.27(**a**) illustrates a naive approach in which each sliver keeps track of all of the up synchrons of the computation. This makes it easy to satisfy the operation order goal, but tail recursion is lost. No pending operations are performed between up synchrons u_2 and u_3 or between u_3 and u_4, so no space should be allocated for these two locations. However, the up synchrons themselves (and the pending management operations needed to process them) take up space!

Figure 5.27(**b**) illustrates an alternate approach, in which each sliver locally keeps track of only those up synchrons immediately below a pending accumulation. This satisfies the

[11]The configuration depicted in the figure can also arise for reasons other than filtering. However, in practice, the problem is most commonly exhibited due to filtering.

tail recursion goal (no space is needed for u_3 and u_4), but operation order is lost. Since the `f` sliver does not have a handle on u_2, nothing forces `f` to be performed before `g`. The problem is that local tail calls do not keep track of global ordering information.

Figure 5.27(**c**) depicts an approach that satisfies both goals. For each pending accumulation, a sliver locally keeps track of the up synchrons both immediately above and below it. This bounds the time in which the accumulation must take place. In addition, a global precedence ordering (arrows labelled P) is maintained between all the up synchrons that forces the rendezvous on them to occur in the expected order. The synchron *precede* operator is used to specify this ordering. This approach is able to eliminate up synchron u_3 but not u_4. However, the space associated with u_4 can be charged to the accumulation `f` in such a way that the space required by *up* phase is proportional to the number of accumulations that need to be performed.

It is challenging to implement approach (**c**) in such a way that the management of synchron precedence doesn't destroy the desired space behavior. The discussion of the formal semantics of synchrons in Section 8.2.2 explains how this can be achieved.

Chapter 6

SYNAPSE: Programming with Slivers and Slags

SYNAPSE[1] is a language that provides a collection of operators (slivers) for manipulating synchronized lazy lists and trees (slags). The language shows the utility of slivers and slags while suppressing their implementation details. While SYNAPSE does not exhibit the full range of expressiveness implied by the sliver technique, it is rich enough for investigating the power and limitations of slivers.

According to Abelson and Sussman, a language has three parts: primitives, means of combination, and means of abstraction [ASS85]. For SYNAPSE, the primitives are a set of higher-order procedures that manipulate synchronized lazy lists and trees. These procedures are carefully implemented so that they obey all of the sliver requirements discussed in Section 5.4.2. The means of combination and means of abstraction for synapse are simply inherited from OPERA, the dialect of Scheme in which SYNAPSE is embedded. Thus, the SYNAPSE "language" is just a library of OPERA procedures.

Ideally, we would like to characterize synchronized lazy lists and trees as abstract data types. An *abstract data type* is a method of defining a data structure in terms of the functions that manipulate it. For example, a pair data type can be defined in terms of a constructor `cons` that builds a pair from two components; selectors `car` and `cdr` that

[1] *SYN*chronized *A*ggregate *P*rogramming *S*tructures *E*nvironment.

extract the left and right elements of a pair; and a *pair?* predicate that determines whether a given entity is a pair. The functions defining an abstract data type are usually required to satisfy certain relationships. For example, for any version of pairing it should be the case that `(car (cons` A B`))` is equivalent to A, `(cdr (cons` A B`))` is equivalent to B, and `(pair? (cons` A B`))` is equivalent to `#t`.

Unfortunately, the *functional* nature of abstract data types is at odds with the *operational* nature of synchronized lazy aggregates. Because their operational handling is intricately intertwined with their structure, synchronized lazy lists and trees resist being characterized as abstract data types. I have been unable to distill the essence of slags into an concise axiomatic theory. Instead, I have taken a more experimental approach. I have developed a suite of higher order slag procedures (slivers) that have proven useful in a wide variety of simple applications. Below, I describe these slivers and illustrate their use.

All of the examples described here are written in OPERA, a dialect of Scheme supporting the concurrency, synchronization, and laziness features motivated in Section 5.1.3. For the most part, it is not necessary to understand the details of OPERA in order to appreciate the examples; for this reason, the discussion of OPERA is deferred until Chapter 7. The one detail that *is* important to the present discussion is that the default evaluation strategy of OPERA evaluates all subexpressions of a procedure call in parallel.[2] This parallel evaluation strategy accounts for the concurrency exhibited by the examples.

6.1 Linear Computations

SYNAPSE supports linear computations through a suite of slivers (procedures) that manipulate synquences (synchronized lazy lists). The core procedures are summarized in Figures 6.1 and 6.2. The core is neither minimal (some slivers can be defined in terms of the others) nor complete (not all useful synquence slivers can be defined in terms of the core). Rather, the core is a set of standard slivers that are useful in many situations. The higher order nature of many of the slivers (i.e., they accept functional arguments) allows them to be tailored to a wide variety of problems. The choice of core procedures was influenced by the

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

list operations of Lisp [Ste90, CR+91] and Haskell[3] [HJW+92], the array operations of APL [Ive87], the vector operations of FX [GJSO92], and Waters's series operations [Wat90].

In the remainder of this section, we will employ the core procedures to illustrate various properties of sliver decomposition within SYNAPSE.

6.1.1 Iteration vs. Recursion

Using the slivers in Figures 6.1 and 6.2, it is easy to define both iterative and recursive procedures in a modular fashion. A combination of slivers behaves iteratively if all of its components have *down* or *across* shape. If any component has *up* shape, the computation may behave recursively. We now consider a series of examples that illustrate the basic behavior of slivers.

Factorial

We begin with the time worn, but still trusty, factorial function. Here are iterative and recursive versions of a factorial procedure expressed in SYNAPSE:

```
(define (fact-iter num)
  (downQ 1 * (genQ num -1+ zero?)))

(define (fact-rec num)
  (upQ 1 * (genQ num -1+ zero?)))
```

Both definitions use a generator that delivers the numbers from the input down to (but not including) zero. The definitions differ only in the shape of the product accumulator.

The iterative nature of `fact-iter` is demonstrated by the following trace of a sample `fact-iter` application:[4]

[3]It is worth noting that SYNAPSE's `upQ` and `up-scanQ` are equivalent to Haskell's `foldr` and `scanr`, but that SYNAPSE's `downQ` and `down-scanQ` are *not* equivalent to Haskell's `foldl` and `scanl`. In particular, the *op* parameter to `downQ` and `down-scanQ` takes its arguments in the opposite order from the corresponding parameter of the Haskell functions. The reason for the discrepency is that the higher order procedural arguments to the SYNAPSE procedures all take their arguments in the same order; this facilitates changing between *down* and *up* accumulators and scanners. The argument order for the higher order procedure in Haskell are intended to faciliate transformations and proofs.

[4]This and all subsequent traces were automatically produced verbatim by the OPERA interpreter. In some cases, illustrating certain behavior required tuning the relative probabilities of particular operators before running the example. (See the discussion of task selection strategies in Section 9.1.1.) Different examples were executed with different tuning parameters. Tuning affects only the probability of observing a particular trace; it does not change the set of possible traces.

Generators:

`(genQ` *init next done?*`)` [*down*]

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

`(produceQ` *initial-state handler*`)` [*down*]

Generates a synquence whose element values are computed from a "state" that is passed down the synquence in a *down* fashion. At every synquence node, *handler* is applied to three arguments: a current state, a binary *yielding* procedure, and a nullary *terminating* procedure. Calling the yielding procedure on a value v and a new state s produces v as the current synquence element and uses s as the next state. Calling the terminating procedure terminates the synquence.

Reducers:

`(downQ` *init op synq*`)` [*down*]

Returns the result of iteratively accumulating the elements of *synq* using the combiner *op* and the initial value *init*. If *synq* has elements $e_1 \ldots e_n$, `downQ` computes:

$$(op\ e_n\ (\ \ldots (op\ e_2\ (op\ e_1\ init)\ \ldots)))$$

`(upQ` *init op synq*`)` [*up*]

Returns the result of recursively accumulating the elements of *synq* using the combiner *op* and the initial value *init*. If *synq* has elements $e_1 \ldots e_n$, `upQ` computes:

$$(op\ e_1\ (op\ e_2 \ldots (op\ e_n\ init)\ \ldots))$$

`(glom-stream` *synq*`)` [*down*]

Returns an OPERA stream of the non-gap elements iteratively collected from *synq*. (Note: this cannot be defined in terms of `downQ`.)

`(down-nthQ` *index synq*`)` [*down*]

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

`(up-nthQ` *index synq*`)` [*up*]

Returns the result of recursively calculating the *index*th element (0-based) of a synquence. An error is signalled if *index* is greater than the length of the synquence.

`(down-lastQ` *index synq*`)` [*down*]

Returns the last (non-gap) element of a synquence. An error is signalled if *synq* contains no elements.

`(up-firstQ` *index synq*`)` [*up*]

Returns the first (non-gap) element of a synquence. An error is signalled if *synq* contains no elements.

Figure 6.1: A summary of SYNAPSE's core synquence generating and reducing slivers. The shape of each sliver appears in brackets to the right of each form.

Transducers:

(`mapQ` *fun synq*) [*across*]
Returns the synquence resulting from the elementwise application of *fun* to *synq*.

(`map2Q` *fun synq1 synq2*) [*across*]
Returns the synquence resulting from the elementwise application of *fun* to corresponding elements of *synq1* and *synq2*. An output element is a gap if either input element is a gap. The length of the output synquence is the shorter of the length of the two input synquences.

(`filterQ` *pred synq*) [*across*]
Returns a synquence with the same structure as *synq* in which every element satisfying *pred* is mapped to itself and all other elements are mapped to gaps.

(`reifyQ` *obj synq*) [*across*]
Returns a synquence that maps every gap of *synq* to *obj* and every ungapped element to itself.

(`unreifyQ` *obj synq*) [*across*]
Returns a synquence that maps every instance of *obj* in *synq* to a gap and every other element to itself.

(`down-scanQ` *init op synq*) [*down*]
Returns the synquence of intermediate accumulated values in the iterative accumulation of *synq* using combiner *op* and initial value *init*.

(`up-scanQ` *init op synq*) [*up*]
Returns the synquence of intermediate accumulated values in the recursive accumulation of *synq* using combiner *op* and initial value *init*.

(`truncateQe` *pred synq*) [*down*]
Returns the prefix of *synq* up to, but not including, the first element for which *pred* is true. (The *e* at the end of the name stands for "exclusive".)

(`truncateQi` *pred synq*) [*down*]
Returns the prefix of *synq* up to, and including, the first element for which *pred* is true. (The *i* at the end of the name stands for "inclusive".)

(`prefixQ` *len synq*) [*down*]
Returns a synquence consisting of the first *len* elements of *synq*. An error is signalled if *len* is greater than the length of *synq*

(`down-shiftQ` *init synq*) [*down*]
Returns a synquence with the same size and gap locations as *synq*, but in which every element has been shifted down to the next ungapped element location. The resulting synquence uses *init* for the first ungapped element location.

(`up-shiftQ` *init synq*) [*up*]
Returns a synquence with the same size and gap locations as *synq*, but in which every element has been shifted up to the next ungapped element location. The resulting synquence uses *init* for the last ungapped element location.

(`appendQ` *synq1 synq2*) [*down*]
Returns a synquence formed by appending the elements of *synq1* to those of *synq2*.

Figure 6.2: A summary of SYNAPSE's core synquence transducing slivers.

```
OPERA> (fact-iter 3)
-----------------------:down[A,0]
(zero? 3) --> ()
(* 3 1) --> 3
(-1+ 3) --> 2
-----------------------:down[A,1]
(zero? 2) --> ()
(-1+ 2) --> 1
(* 2 3) --> 6
-----------------------:down[A,2]
(zero? 1) --> ()
(* 1 6) --> 6
(-1+ 1) --> 0
-----------------------:down[A,3]
(zero? 0) --> #t
; Value: 6
```

The trace begins with the expression and ends with its value. The intervening lines of text correspond to various events in the program's execution. Lines of the form

(*primop arg1* ... *argn*) `-->` *result*

are *primop events* that indicate the performance of the primitive operation *primop* on the listed args.[5] A dotted line is a *barrier event* that represents a rendezvous of all the slivers in a lock step component at the down or up barrier of a barrion. In this case, the fact that all dotted lines are labelled `:down` indicates that the computation is iterative — it never returns through any up barriers. The information following the `:down` label (e.g., `[A,2]`) consists of a *generator label* (`A`) and a *barrion index* (`2`). All barrions generated by the same call to `genQ` share the same generator label, but have a unique barrion index. These automatically generated pieces of information make it easier for humans to parse complex traces into meaningful parts.

Barrier events partition traces into *episodes* of primop events. In the `fact-iter` example, each episode contains a zero test and decrement from the `genQ` sliver and a multiplication from the `downQ` sliver.[6] The interleaving of primitive operations from `genQ` and `downQ` illustrates the concurrent evaluation that is fundamental to the sliver technique.

[5]Only operators that have been declared *traceable* produce printed operator events. SYNAPSE has a mechanism for specifying which operators are traceable. In the examples, only the "interesting" operators are traced.

[6]The `zero?` procedure is always performed first, because the other operations will only be performed if the test fails. However, as suggested by the trace, the `*` and `-1+` are unrelated by data dependence and may be performed in either order.

The barriers are important in this example because they they guarantee that `fact-iter` behaves like a monolithic iterative factorial procedure even though it is constructed out of two slivers. The corresponding monolithic factorial could be written as:

```
(define (fact-iter-mono num)
  (define (loop n acc)
    (if (zero? n)
        acc
        (loop (-1+ n) (* n acc))))
  (loop num 1))
```

Each of the barriers depicted in the `fact-iter` trace corresponds to one of the calls to `loop` in the computation of `(fact-iter-mono 3)`. Except for the absence of dotted lines, the following sample trace of `fact-iter-mono` is indistinguishable from a possible trace of `fact-iter`:

```
OPERA> (fact-iter-mono 3)
(zero? 3) --> ()
(-1+ 3) --> 2
(* 3 1) --> 3
(zero? 2) --> ()
(-1+ 2) --> 1
(* 2 3) --> 6
(zero? 1) --> ()
(* 1 6) --> 6
(-1+ 1) --> 0
(zero? 0) --> #t
; Value: 6
```

This is not just a lucky coincidence. Barriers *guarantee* that the possible traces of `fact-iter` on any given input are *exactly* the same as the possible traces of `fact-iter-mono`. In fact, the whole purpose of the barriers is to simulate the behavior of strict, monolithic procedure calls in a computation distributed over several slivers.

The `fact-rec` procedure gives rise to a computation with a very different trace:

```
OPERA> (fact-rec 3)
------------------------:down[A,0]
(zero? 3) --> ()
(-1+ 3) --> 2
------------------------:down[A,1]
(zero? 2) --> ()
(-1+ 2) --> 1
------------------------:down[A,2]
(zero? 1) --> ()
(-1+ 1) --> 0
------------------------:down[A,3]
(zero? 0) --> #t
------------------------:up[A,3]
(* 1 1) --> 1
------------------------:up[A,2]
(* 2 1) --> 2
------------------------:up[A,1]
(* 3 2) --> 6
------------------------:up[A,0]
; Value: 6
```

The presence of up barriers indicates that the computation is not an iteration, but a general recursion. Zero tests and decrements are still performed in the *down* phase (because `genQ` has *down* shape), but all the multiplications are performed in the *up* phase – exactly what we expect for a recursive factorial computation. The barrion indices help us match an up barrier with the corresponding down barrier; barriers with the same index are from the same barrion.

The components of the factorial procedures are likely to be used again, so it is a good idea to name them:

```
(define (to-1 num) (genq num -1+ zero?))

(define (down-* synq) (downQ 1 * synq))

(define (up-* synq) (upQ 1 * synq))
```

The above factorial procedures can be rewritten in terms of these new abstractions without changing the behavior exhibited by the traces:

```
(define (fact-iter num) (down-* (to-1 num)))

(define (fact-rec num) (up-* (to-1 num)))
```

It is also possible to represent factorial procedures that count up towards the input number rather than counting down from it:

```
(define (fact-iter2 num) (down-* (from-to 1 num)))

(define (fact-rec2 num) (up-* (from-to 1 num)))

(define (from-to start stop)
  (genQ start 1+ (lambda (n) (> n stop))))
```

These versions *would* yield traces different from the ones illustrated above.

Sum of Squares

As another simple example, consider SYNAPSE programs for summing the squares of the elements in a list:

```
(define (sum-squares-iter lst)
  (down-+ (mapQ square (splay-list lst))))

(define (sum-squares-rec lst)
  (up-+ (mapQ square (splay-list lst))))

;; Utilities
(define (splay-list lst)
  (mapQ car (splay-sublists lst)))

(define (splay-sublists lst)
  (genQ lst cdr null?))

(define (down-+ synq) (downQ 0 + synq))

(define (up-+ synq) (upQ 0 + synq))
```

Here, `splay-sublists` generates the successive sublists of a list, while `splay-list` generates the successive elements of a list. The two versions of the summing program are identical except for the shape of the accumulator.

Sample traces for these two programs appear in Figure 6.3. As expected, the first trace shows how shared barriers and *down*-shaped slivers force `sum-squares-iter` to behave as if it were a monolithic loop. In the second trace, the `+` operations in the *up* phase are expected, but `car` and `square` operations in the *up* phase might seem a little surprising. Since both of these are mapping operations that occur between a *down* sliver and an *up* sliver, they have an *across* orientation (see Section 5.2.3). Recall that it is permissible to perform an *across* operator any time after call initiation but before call return. The sample trace obeys these constraints.

```
OPERA> (sum-squares-iter '(2 3 5))
-----------------------:down[A,0]
(null? (2 3 5)) --> ()
(cdr (2 3 5)) --> (3 5)
(car (2 3 5)) --> 2
(square 2) --> 4
(+ 4 0) --> 4
-----------------------:down[A,1]
(null? (3 5)) --> ()
(cdr (3 5)) --> (5)
(car (3 5)) --> 3
(square 3) --> 9
(+ 9 4) --> 13
-----------------------:down[A,2]
(null? (5)) --> ()
(cdr (5)) --> ()
(car (5)) --> 5
(square 5) --> 25
(+ 25 13) --> 38
-----------------------:down[A,3]
(null? ()) --> #t
; Value: 38


OPERA> (sum-squares-rec '(2 3 5))
-----------------------:down[A,0]
(null? (2 3 5)) --> ()
(cdr (2 3 5)) --> (3 5)
-----------------------:down[A,1]
(null? (3 5)) --> ()
(cdr (3 5)) --> (5)
-----------------------:down[A,2]
(null? (5)) --> ()
(cdr (5)) --> ()
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(car (2 3 5)) --> 2
(square 2) --> 4
(car (5)) --> 5
(square 5) --> 25
(+ 25 0) --> 25
-----------------------:up[A,2]
(car (3 5)) --> 3
(square 3) --> 9
(+ 9 25) --> 34
-----------------------:up[A,1]
(+ 4 34) --> 38
-----------------------:up[A,0] ; Value: 38
```

Figure 6.3: Traces of iterative and recursive programs for summing the squares of elements in a list.

This example underscores that `mapQ` behaves differently in different contexts. In general, this flexibility enhances the reusability of the `mapQ` sliver. (See Section 6.1.3 for specific examples.) However, in the case of `sum-squares-rec`, the fact that *across* operators can exhibit *up* behavior may be undesirable. Delaying the `car` operators until the *up* phase means that `sum-square-rec` maintains a pointer to the list argument until it returns a final result, which can unnecessarily prevent the list cells from being garbage collected sooner.

The default behavior can be overridden by explicitly specifying that the `car`s performed by `splay-list` should be *down* operators. The following `force-down` sliver uses the data dependencies implicit in `down-scanQ` to force all elements of a synquence to be computed in the *down* phase:

```
(define (force-down synq)
  (down-scanQ 'ignore
              (lambda (current previous) current)
              synq))
```

Using `force-down`, it is easy to rewrite `splay-list` so that it always performs `car` operations as early as possible, rather than as late as possible:

```
(define (splay-list lst)
  (force-down (mapq car (splay-sublists lst))))
```

The post-modification trace in Figure 6.4 shows that the change has the desired effect. A similar change could force the multiplications to happen in the *down* phase as well. The moral of this example is that slivers permit control aspects of a program to be fine-tuned in a modular fashion.

List Accumulation

It is instructive to consider computations where the direction of accumulation makes a difference to the final answer. Here are down and up accumulators for pairing, the canonical non-associative, non-commutative binary operator:[7]

```
(define (down-cons synq) (downQ '() tcons synq))

(define (up-cons synq) (upQ '() tcons synq))
```

[7] `Tcons` is a variant of the usual pairing procedure, `cons`. It is use here for a technical reason explained in Section 6.1.3.

```
OPERA> (sum-squares-rec '(2 3 5))
-----------------------:down[A,0]
(null? (2 3 5)) --> ()
(cdr (2 3 5)) --> (3 5)
(car (2 3 5)) --> 2
-----------------------:down[A,1]
(null? (3 5)) --> ()
(car (3 5)) --> 3
(cdr (3 5)) --> (5)
-----------------------:down[A,2]
(null? (5)) --> ()
(cdr (5)) --> ()
(car (5)) --> 5
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(* 5 5) --> 25
(+ 25 0) --> 25
-----------------------:up[A,2]
(* 3 3) --> 9
(+ 9 25) --> 34
-----------------------:up[A,1]
(* 2 2) --> 4
(+ 4 34) --> 38
-----------------------:up[A,0]
; Value: 38
```

Figure 6.4: Sample trace of `sum-squares-rec` after `splay-list` has been modified to perform all `car`s in the *down* phase.

Figure 6.5 shows traces of computations in which these slivers are applied to the synquence produced by `(from-to 1 3)`. The traces show that `up-cons` returns a list of the synquence elements in the order in which they were generated (smallest to largest), while `down-cons` returns the elements in the reverse order. As indicated by the down and up barriers, the order-preserving version requires a linear control stack, while the order-reversing version uses only constant control space.

The fact that `up-cons` preserves the order of elements is one of the main motivations to use recursion (vs. iteration) in procedures that produce lists. One alternate approach to preserving element order is to use `down-cons` to produce the list, and then iteratively reverse this result. Another alternative, as discussed in Section 2.1.5, is to employ a `cdr`-bashing technique that maintains the element order *and* uses constant control space.[8] Figure 6.6 shows the definition a `cdr`-bashing pairing accumulator and a sample trace of its execution. The ability to code `down-cons!` as an independent sliver that has the "right" behavior when composed with other slivers illustrates the power of SYNAPSE.

Iterated Scans

We conclude this section with a less trivial example. The problem is to compute from a list its *nth down* (or *up*) *scan*. The zeroth scan of a list is the list itself; the first scan is a list of the intermediate results from accumulating the list in the specified direction; the second scan is the result of scanning the first scan; and so on. The goal is to write general nth scanning procedures that simulate the behavior of individually crafted monolithic verions for each n.

Figure 6.7 gives sliver-based solutions to the problem. The solutions employ a higher-order `repeated` procedure that returns the n-fold composition of a given procedure `f`. As indicated by the sample traces in Figures 6.8 and 6.9, successive scans are synchronized with each other even though they are dynamically glued together by the higher-order `repeated` procedure. Note how the use of `down-cons!` ensures that `nth-down-scan` generates an

[8]Waters has recently shown that the reversal technique and the `cdr`-bashing technique are practically indistinguishable in terms of efficiency [Wat]. In light of this result, he argues for using the simpler one — i.e., the reversal approach. However, in a language like SYNAPSE, where the `cdr`-bashing technique can be expressed as a modular component, `cdr`-bashing may actually the "simpler" approach.

```
OPERA> (down-cons (from-to 1 3))
-----------------------:down[A,0]
(> 1 3) --> ()
(1+ 1) --> 2
(tcons 1 ()) --> (1)
-----------------------:down[A,1]
(> 2 3) --> ()
(1+ 2) --> 3
(tcons 2 (1)) --> (2 1)
-----------------------:down[A,2]
(> 3 3) --> ()
(1+ 3) --> 4
(tcons 3 (2 1)) --> (3 2 1)
-----------------------:down[A,3]
(> 4 3) --> #t
; Value: (3 2 1)

OPERA> (up-cons (from-to 1 3))
-----------------------:down[A,0]
(> 1 3) --> ()
(1+ 1) --> 2
-----------------------:down[A,1]
(> 2 3) --> ()
(1+ 2) --> 3
-----------------------:down[A,2]
(> 3 3) --> ()
(1+ 3) --> 4
-----------------------:down[A,3]
(> 4 3) --> #t
-----------------------:up[A,3]
(tcons 3 ()) --> (3)
-----------------------:up[A,2]
(tcons 2 (3)) --> (2 3)
-----------------------:up[A,1]
(tcons 1 (2 3)) --> (1 2 3)
-----------------------:up[A,0]
; Value: (1 2 3)
```

Figure 6.5: Sample `down` and `up` accumulations of a pairing operator.

```
(define (down-cons! default-cdr synq)
  (let ((dummy (cons 'ignore default-cdr)))
    (begin
      (downQ dummy
             (lambda (val pair)
               (let ((new (tcons val default-cdr)))
                 (begin
                   (set-cdr! pair new)
                   new)))
             synq)
      (cdr dummy))))
```

```
OPERA> (down-cons! '() (from-to 1 3))
(cons ignore ()) --> (ignore)
-----------------------:down[A,0]
(> 1 3) --> ()
(1+ 1) --> 2
(tcons 1 ()) --> (1)
(set-cdr! (ignore) (1)) --> #t
-----------------------:down[A,1]
(> 2 3) --> ()
(1+ 2) --> 3
(tcons 2 ()) --> (2)
(set-cdr! (1) (2)) --> #t
-----------------------:down[A,2]
(> 3 3) --> ()
(1+ 3) --> 4
(tcons 3 ()) --> (3)
(set-cdr! (2) (3)) --> #t
-----------------------:down[A,3]
(> 4 3) --> #t
(cdr (ignore 1 2 3)) --> (1 2 3)
; Value: (1 2 3)
```

Figure 6.6: A `cdr`-bashing list accumulator, `down-cons!`, along with a sample execution trace.

```
(define (nth-down-scan num init combiner list)
  (down-cons! ((nth-down-scanQ num init combiner) (splay-list list))))

(define (nth-down-scanQ num init combiner)
  (repeated (lambda (synq) (down-scanQ init combiner synq))
            num))

(define (nth-up-scan num init combiner list)
  (up-cons ((nth-up-scanQ num init combiner) (splay-list list))))

(define (nth-up-scanQ num init combiner)
  (repeated (lambda (synq) (up-scanQ init combiner synq))
            num))

(define (repeated f n)
  (if (= n 0)
      identity
      (compose f (repeated f (- n 1)))))

(define (compose f g)
  (lambda (x) (f (g x))))

(define (identity x) x)
```

Figure 6.7: Sliver-based code for computing the nth iterated down and up scans of a list.

iterative computation. For `nth-up-scan`, the *up* orientation of the accumulation operations requires an *up* list collector (`up-cons`).

6.1.2 Expressive Power

Before we investigate more advanced synquence features, it is worthwhile to step back for a moment and comment on how slivers and synquences enrich the expressive power of a language. Consider the sliver implementations of some standard list utilities (see Figure 6.10). It may seem somewhat odd to convert lists into synquences just to perform some simple synquence manipulations. And in fact, in straightforward implementations of slivers, synquence manipulation would incur sufficient overhead to discourage programmers from actually writing the list utilities as indicated in Figure 6.10. However, as I will argue later, it is likely that techniques similar to those used by Waters in his series package [Wat91] can be used to compile such definitions into efficient code. More important, the kinds of decompositions shown in Figure 6.10 have many expressiveness advantages that transcend issues of efficiency:

- Synquences are a common currency in which to express a wide range of linear iterations and recursions. Using a standard collection of synquence slivers in conjunction with conversion routines makes it unnecessary to reimplement common functionality for different datatypes. For example, mapping a function over a vector can be expressed by sandwiching `mapQ` between vector analogs of `splay-list` and `up-cons`.

- Synchronized lazy lists are a more suitable common currency than other standard linear structures (lists, vectors, files, lazy lists) for expressing operational notions like lock step processing and the distinction between iteration and recursion. The synchronized nature of synquences guarantees that each of the modular procedures defined in Figure 6.10 behaves like the standard monolithic implementations. Furthermore, the use of `downQ` in the definitions of `reverse` and `reverse!` indicates that these generate iterative processes, while the `upQ` in the other definitions indicates recursive processes.

- Sliver-based decompositions highlight the structural similarities and differences between procedures. For example, Figure 6.10 suggests insights like the following:

```
OPERA> (nth-down-scan 4 0 + '(1 20 300))
(= 4 0) --> ()
(- 4 1) --> 3
(= 3 0) --> ()
(- 3 1) --> 2
(= 2 0) --> ()
(- 2 1) --> 1
(= 1 0) --> ()
(- 1 1) --> 0
(= 0 0) --> #t
(tcons ignore ()) --> (ignore)
----------------------:down[A,0]
(null? (1 20 300)) --> ()
(car (1 20 300)) --> 1
(cdr (1 20 300)) --> (20 300)
(+ 1 0) --> 1
(+ 1 0) --> 1
(+ 1 0) --> 1
(+ 1 0) --> 1
(tcons 1 ()) --> (1)
(set-cdr! (ignore) (1)) --> #t
----------------------:down[A,1]
(null? (20 300)) --> ()
(car (20 300)) --> 20
(+ 20 1) --> 21
(+ 21 1) --> 22
(+ 22 1) --> 23
(+ 23 1) --> 24
(cdr (20 300)) --> (300)
(tcons 24 ()) --> (24)
(set-cdr! (1) (24)) --> #t
----------------------:down[A,2]
(null? (300)) --> ()
(car (300)) --> 300
(+ 300 21) --> 321
(+ 321 22) --> 343
(+ 343 23) --> 366
(+ 366 24) --> 390
(cdr (300)) --> ()
(tcons 390 ()) --> (390)
(set-cdr! (24) (390)) --> #t
----------------------:down[A,3]
(null? ()) --> #t
(cdr (ignore 1 24 390)) --> (1 24 390)
; Value: (1 24 390)
```

Figure 6.8: Sample trace of the fourth additive iterative scan of the list `(1 20 300)`.

```
OPERA> (nth-up-scan 4 0 + '(1 20 300))
(= 4 0) --> ()
(- 4 1) --> 3
(= 3 0) --> ()
(- 3 1) --> 2
(= 2 0) --> ()
(- 2 1) --> 1
(= 1 0) --> ()
(- 1 1) --> 0
(= 0 0) --> #t
-----------------------:down[A,0]
(null? (1 20 300)) --> ()
(cdr (1 20 300)) --> (20 300)
(car (1 20 300)) --> 1
-----------------------:down[A,1]
(null? (20 300)) --> ()
(cdr (20 300)) --> (300)
(car (20 300)) --> 20
-----------------------:down[A,2]
(null? (300)) --> ()
(cdr (300)) --> ()
(car (300)) --> 300
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(+ 300 0) --> 300
(+ 300 0) --> 300
(+ 300 0) --> 300
(+ 300 0) --> 300
(tcons 300 ()) --> (300)
-----------------------:up[A,2]
(+ 20 300) --> 320
(+ 320 300) --> 620
(+ 620 300) --> 920
(+ 920 300) --> 1220
(tcons 1220 (300)) --> (1220 300)
-----------------------:up[A,1]
(+ 1 320) --> 321
(+ 321 620) --> 941
(+ 941 920) --> 1861
(+ 1861 1220) --> 3081
(tcons 3081 (1220 300)) --> (3081 1220 300)
-----------------------:up[A,0]
; Value: (3081 1220 300)
```

Figure 6.9: Sample trace of the fourth additive iterative scan of the list `(1 20 300)`.

```
;;; A recursive computation of the length of a list
(define (length lst)
  (upQ 0
       (lambda (ignore count) (1+ count))
       (splay-list lst)))

;;; A recursive computation that returns a list containing (in order) the
;;; elements of the two given lists.  LST1 is copied; LST2 is not.
(define (append lst1 lst2)
  (upQ lst2 tcons (splay-list lst1)))

;;; A recursive computation returning a new list with the same elements
;;; as the given one.
(define (copy lst)
  (up-cons (splay-list lst)))

;;; An iterative computation that returns a new list whose
;;; elements are in reverse order from the given list
(define (reverse lst)
  (down-cons (splay-list lst)))

;;; An iterative computation that performs a destructive
;;; reversal of the pairs in a given list.
(define (reverse! lst)
  (downQ '()
         (lambda (pair prev)
           (set-cdr! pair prev)
           pair)
        (splay-sublists lst)))

;;; An iterative computation that returns the last pair of LST; or nil
;;; if it's empty
(define (last-pair lst)
  (downQ '()
         (lambda (pair prev) pair)
         (splay-sublists lst)))

;;; A recursive computation that returns a new list each of whose elements
;;; is the result of applying F to the corresponding element of the original.
(define (map f lst)
  (up-cons (mapQ f (splay-list lst))))

;;; A recursive computation that returns a new list each of whose elements
;;; is the result of applying F to the corresponding sublist of the original.
(define (map-list f lst)
  (up-cons (mapQ f (splay-sublists lst))))
```

Figure 6.10: Expressing some standard list utilities in terms of slivers.

- `Append`, `copy`, and `reverse` are all cons accumulations on a list; `copy` differs from `reverse` only in the direction of accumulation, while it differs from `append` only in the initial value.
- `Reverse!` and `last-pair` procedures are surprisingly similar; both iterate over the sublists of the given list and return the last pair, but `reverse!` additionally resets `cdr` pointers on the way down.
- `Map` can naturally be viewed as a generalization of `copy`.
- `Map` and `map-list` differ only in the choice of generator.

- Expressing procedures in terms of slivers suggests alternate ways to implement a procedure. For example, various classes of `up` accumulations can be changed to `down` accumulations. For any binary accumulator f such that $(fa(fbc)) = (fb(fac))$ for all a, b, and c, it is always possible to replace `upQ` by `downQ`. In the `length` procedure, the function denoted by `(lambda (elt count) (1+ count))` satisfies this condition, so `length` can be changed to an iterative procedure by substituting `downQ` for `upQ`. Another valid recursive to iterative transformation is to use the `cdr`-bashing trick to replace `(upQ` *init* `tcons` *synq*`)` by `(down-cons!` *init* *synq*`)`. This technique could be used to make iterative versions of `append`, `copy`, `map`, and `map-list`. A host of other transformations from the literature on program transformation [DR76, Dar82, Bac78, Bel86, Bir89a, Bir86, Bir88, Coh83, Str71, WS73, Pet84] find a convenient expression in terms of slivers.

The above points indicate that, regardless of their appropriateness as an implementation language, slivers are a powerful way to design, specify, reason about, and explain programs. They form a basis of "subprocedural particles" into which many common procedures can be easily decomposed; as such, they have excellent prospects for reusability. Although other aggregate structures share many of these advantages, slivers additionally allow the programmer to express finer-grained control over the resultant computation when it's important to do so. In this sense, it is possible to view the definitions in Figure 6.10 as specifications for the operational behavior of the procedures; that they are executable specifications is an added bonus.

6.1.3 Laziness

In the case of synquences, the 'lazy" in "synchronized lazy aggregates" means that neither the element nor the tail of a synquence is computed until it is actually needed. (In comparison, Scheme streams have lazy tails but not lazy elements.) Additionally, both elements and tails are *memoized* [ASS85, Hug85] so that they are only computed once if they are computed at all.

Laziness leads to two important features of synquences:

1. Synquences may be conceptually infinite.

2. An element is not computed if it never used.

Infinite Synquences

An easy way to generate an infinite synquence is to use `genQ` with a *done?* predicate that is never true. The following sliver, which produces the successive powers of a base (starting with the zeroth power), is based on this idea:

```
(define (powers base)
  (genQ 1
        (lambda (prev) (* prev base))
        (lambda (prev) #f)))
```

For example, the expression `(powers 3)` produces a conceptually infinite synquence with elements 1, 3, 9, 27, 81, ….

In order to observe the initial elements of an infinite synquence, it is handy to define a `down-print` sliver that prints out the successive elements of a synquence:

```
(define (down-print synq)
  (for-eachQ output synq))

(define (for-eachQ proc synq)
  (downQ 'ignore
         (lambda (current ignore) (proc current))
         synq))
```

`For-eachQ` is a higher-order sliver for iteratively performing a given procedure `proc` on every element of a synquence. `Output` is an OPERA primitive for printing a given element on a separate line preceded with `OUTPUT:`. Here is a sample use of `down-print` on the powers-of-three synquence:

```
OPERA> (down-print (powers 3))
-----------------------:down[A,0]
(* 1 3) --> 3
OUTPUT: 1
-----------------------:down[A,1]
OUTPUT: 3
(* 3 3) --> 9
-----------------------:down[A,2]
OUTPUT: 9
(* 9 3) --> 27
-----------------------:down[A,3]
OUTPUT: 27
(* 27 3) --> 81
-----------------------:down[A,4]
(* 81 3) --> 243
OUTPUT: 81
-----------------------:down[A,5]
(* 243 3) --> 729
           ⋮
```

Within each episode, the current power is printed and the next one is computed (in some arbitrary order).

Most practical manipulation of infinite synquences requires some means of truncating the synquence or selecting elements from it. Here are some simple examples of infinite synquence manipulation using some of the truncaters and selectors defined in Figures 6.1 and 6.2 (tracing has been turned off in these examples):

```
OPERA> (down-+ (truncateQe (lambda (n) (> n 10)) (powers 3)))
; Value: 13

OPERA> (down-+ (truncateQi (lambda (n) (> n 10)) (powers 3)))
; Value: 40

OPERA> (down-+ (prefixQ 4 (powers 2)))
; Value: 15

OPERA> (down-nthQ 10 (powers 2))
; Value: 1024
```

Infinite synquences can help us use the `nth-down-scanQ` procedure from Section 6.1.1 to compute Pascal's triangle. As depicted in Figure 6.11, the elements of Pascal's triangle can be generated as the intermediate results of iteratively scanning an infinite sequence of ones. Here is a procedure that produces the element at a given position in the array of elements suggested by the figure:

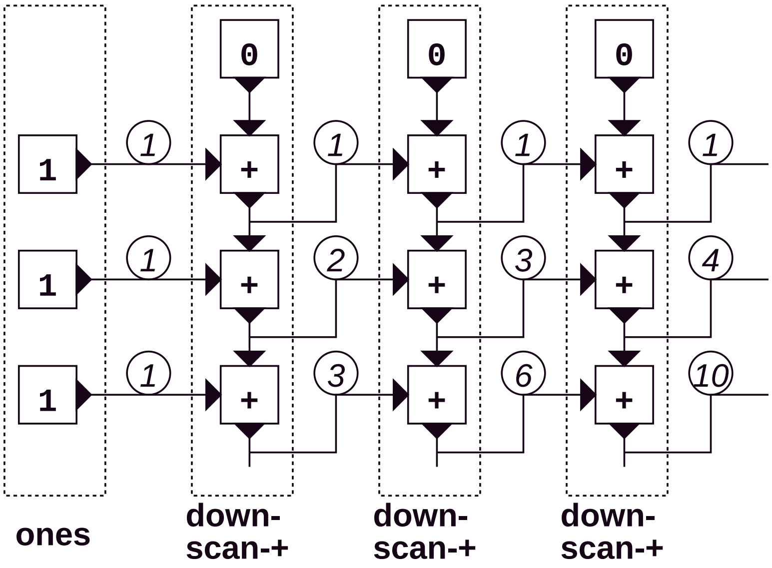


Figure 6.11: One way to organize the computation of the elements of Pascal's triangle. The circled numbers are the computed elements.

```
(define (pascal row column)
  ;; ROW and COLUMN are zero-based indices
  (down-nthQ row
             ((nth-down-scanQ column 0 +) (ones))))

(define (ones)
  (genQ 1 (lambda (x) x) (lambda (x) #f)))
```

Lazy Elements

The second feature of laziness (lazy elements) is more subtle. The computation of an element is delayed until it is needed; if it is never required, it is never computed.

Consider the `contrived-sum` sliver in Figure 6.12, which collects the current element if the accumulated value is even but ignores it (and adds 1) if the accumulated value is odd. As shown in the accompanying trace, the squaring operation is not performed on the even numbers generated by `(to-1 5)` because that value is not required by the accumulator. Although the savings is minimal in this case, it can be much larger if the mapped function is expensive to compute.

Lazy elements serve the goal of operational faithfulness by modelling the kinds of con-

```
(define (contrived-sum synq)
  (downQ 0
         (lambda (current sum)
           (if (even? sum)
               (+ current sum)
               (1+ sum)))
         synq))


OPERA> (contrived-sum (mapq square (to-1 5)))
-----------------------:down[A,0]
(zero? 5) --> ()
(-1+ 5) --> 4
(square 5) --> 25
(+ 25 0) --> 25
-----------------------:down[A,1]
(zero? 4) --> ()
(-1+ 4) --> 3
(1+ 25) --> 26
-----------------------:down[A,2]
(zero? 3) --> ()
(-1+ 3) --> 2
(square 3) --> 9
(+ 9 26) --> 35
-----------------------:down[A,3]
(zero? 2) --> ()
(-1+ 2) --> 1
(1+ 35) --> 36
-----------------------:down[A,4]
(zero? 1) --> ()
(-1+ 1) --> 0
(square 1) --> 1
(+ 1 36) --> 37
-----------------------:down[A,5]
(zero? 0) --> #t
; Value: 37
```

Figure 6.12: An example illustrating the laziness of sliver elements. Note that `square` is not performed on even elements.

ditional computations found in monolithic recursions. The above example corresponds to the following monolithic procedure:

```
(define (contrived-sum-mono n)
  (define (loop num sum)
    (if (zero? num)
        sum
        (if (even? sum)
            (loop (-1+ num) (+ (square num) sum))
            (loop (-1+ num) (1+ sum)))))
  (loop n sum))
```

In this procedure, the evaluation of `(square num)` is controlled by the `(even? sum)` test. In conjunction with a demand-driven model of evaluation, lazy elements make it possible to express this conditionalization in a modular fashion.

Sometimes it is desirable to override the default laziness of elements. Consider what happens if we collect the results of a map using `cons`:

```
OPERA> (downQ '() cons (mapQ square (to-1 2)))
-----------------------:down[A,0]
(zero? 2) --> ()
(-1+ 2) --> 1
(cons lazon:50 ()) --> (lazon:50)
-----------------------:down[A,1]
(zero? 1) --> ()
(-1+ 1) --> 0
(cons lazon:51 (lazon:50)) --> (lazon:51 lazon:50)
-----------------------:down[A,2]
(zero? 0) --> #t
; Value: (lazon:51 lazon:50)
```

In OPERA, `cons` does not require the values of its operands. If an operand to `cons` is a lazy element, it appears as a *lazon* object that represents the delayed computation. Each of the lazons in the above example represents a suspended squaring computation. If our goal is to model a loop in which a squaring is performed each iteration, then the above computation fails to meet this goal.

The computation suspended within a lazon is initiated by *touching* the lazon. A lazon is automatically touched if it appears in a context that requires the value of the suspended computation.[9] The `touch` procedure can be used to explicitly touch a lazon. Here is

[9]Automatic touching distinguishes lazons from Scheme's delayed objects [ASS85], whose suspended computations can only be explicitly touched via the `force` procedure.

a modified version of the list collection example in which the first argument to `cons` is explicitly touched:

```
OPERA> (downQ '()
              (lambda (elt lst)
                (cons (touch elt) lst))
              (mapQ square (to-1 2)))
-----------------------:down[A,0]
(zero? 2) --> ()
(-1+ 2) --> 1
(square 2) --> 4
(cons 4 ()) --> (4)
-----------------------:down[A,1]
(zero? 1) --> ()
(-1+ 1) --> 0
(square 1) --> 1
(cons 1 (4)) --> (1 4)
-----------------------:down[A,2]
(zero? 0) --> #t
; Value: (1 4)
```

Because of the `touch`, the squarings are performed in lock step with the creation of the resulting list. The `tcons` procedure introduced in Section 6.1.1 is just a synonym for the `lambda` expression in the above example.

We emphasize that the synquence processing model uses laziness and strictness to handle different kinds of communication between program parts. Communication of arguments and results across the barrier between a sliver call and its subcalls is strict by default. This strictness models the behavior of strict call within a monolithic recursion. Communication of elements between one sliver and another is lazy by default to model the demand-driven computation associated with one layer of a monolithic recursion. In the terminology of Chapter 5, argument/result communication is strict, while consump/product communication is lazy (see Figure 5.4). Since these default mechanisms are not appropriate for every situation, there are mechanisms for overriding the defaults.

6.1.4 Fan-in

The slivers examined up to this point take at most one input synquence. Many situations require slivers that accept multiple input synquences. Such slivers are said to exhibit *fan-in*. An inner product sliver that computes the sum of the pairwise products of two synquences is an example of a sliver with fan-in. According to the barrion preservation requirement (see Section 5.4.2), a sliver with two synquence inputs must somehow combine each pair of corresponding input barrions into a single output barrion. None of the slivers introduced so far has this capability.

Fortunately, a single higher-order sliver is able to handle most forms of fan-in. This sliver, `map2Q`, is just a generalization of `mapQ` that applies a given binary function to the respective elements of *two* input synquences. The length of the result is the length of the shorter input synquence. This property is particularly helpful when one synquence is finite but the other is infinite.

With `map2Q`, an inner product sliver can be defined as:

```
(define (inner-product synq1 synq2)
  (down-+ (map2Q * synq1 synq2)))
```

Figure 6.13 shows an execution trace for an application of `inner-product` in which the first argument is finite and the second is infinite. The unmatched elements of the second synquence are ignored because of the truncation property of `map2Q`.

Even in the presence of fan-in, each epsiode is still framed by a single pair of down barriers. Each of the two generators (`splay-list` and `powers`) creates independent barrions for each synchronization event, but `map2Q` unifies these into a single barrion. The lock step processing of the slivers in the inner product example crucially depends on the sharing of synchronization information entailed by barrion propagation and unification.

Like `mapQ`, `map2Q` has an *across* shape because its behavior is determined by context. In the `inner-product` example, `map2Q` participates in an iteration, but it works in other configurations as well. For example, Figure 6.14 illustrates how the mapped function is forced to happen in the up phase when `map2Q` is wedged between two *up* scanners and an *up* accumulator. It is important to note that truncation property of `map2Q` limits the length of the result synquence, but does not prevent the extra *up* processing that must be done on the longer input synquence. In a monolithic recursion, the same behavior could be obtained by breaking the computation up into two stages: one in which the corresponding elements of both inputs are processed, and one in which the "overflow" elements of the second inputs are processed. The sliver approach is a decidedly simpler way of expressing this sort of complex, staged recursion.

```
OPERA> (inner-product (splay-list '(2 3 4)) (powers 10))
-----------------------:down[A,0]
(null? (2 3 4)) --> ()
(* 1 10) --> 10
(car (2 3 4)) --> 2
(* 2 1) --> 2
(+ 2 0) --> 2
(cdr (2 3 4)) --> (3 4)
-----------------------:down[A,1]
(null? (3 4)) --> ()
(* 10 10) --> 100
(cdr (3 4)) --> (4)
(car (3 4)) --> 3
(* 3 10) --> 30
(+ 30 2) --> 32
-----------------------:down[A,2]
(null? (4)) --> ()
(* 100 10) --> 1000
(car (4)) --> 4
(* 4 100) --> 400
(+ 400 32) --> 432
(cdr (4)) --> ()
-----------------------:down[A,3]
(null? ()) --> #t
; Value: 432
```

Figure 6.13: Sample execution trace of `list-inner-product`, a procedure that obtains fan-in via `map2Q`.

```
OPERA> (up-cons (map2Q max
                       (up-scanQ 0 + (to-1 4))
                       (up-scanQ 1 * (splay-list '(7 5)))))
-----------------------:down[A,0]
(null? (7 5)) --> ()
(zero? 4) --> ()
(-1+ 4) --> 3
(cdr (7 5)) --> (5)
(car (7 5)) --> 7
-----------------------:down[A,1]
(zero? 3) --> ()
(null? (5)) --> ()
(-1+ 3) --> 2
(car (5)) --> 5
(cdr (5)) --> ()
-----------------------:down[A,2]
(null? ()) --> #t
(zero? 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
-----------------------:up[A,4]
(+ 1 0) --> 1
-----------------------:up[A,3]
(+ 2 1) --> 3
-----------------------:up[A,2]
(+ 3 3) --> 6
(* 5 1) --> 5
(max 6 5) --> 6
(tcons 6 ()) --> (6)
-----------------------:up[A,1]
(+ 4 6) --> 10
(* 7 5) --> 35
(max 10 35) --> 35
(tcons 35 (6)) --> (35 6)
-----------------------:up[A,0]
; Value: (35 6)
```

Figure 6.14: An example underscoring the *across* nature of `map2Q`. It is challenging to write a monolithic recursion that exhibits this behavior.

```
;;; Return a synquence of pairs of the corresponding elements of inputs.
;;; This is handy for constructing other slivers with fan-in.
(define (zipQ synq1 synq2)
  (map2Q cons synq1 synq2))

;;; Mapper on three input synquences.
(define (map3Q f synq1 synq2 synq3)
  (map2Q (lambda (elt1 elt2&elt3)
           (f elt1 (car elt2&elt3) (cdr elt2&elt3)))
         synq1
         (zipQ cons synq2 synq3)))

;;; Iterative accumulator over two input synquences. FUN is a ternary function
;;; that maps two elements and an accumulator value to a new accumulator value.
(define (down2Q fun synq1 synq2)
  (downQ (lambda (pair acc)
           (f (car pair) (cdr pair) acc))
         (zipQ synq1 synq2)))
```

Figure 6.15: Many slivers that exhibit fan-in can be defined in terms of `map2Q`.

Figure 6.15 shows how `map2Q` can be used to construct some other slivers that accept multiple synquence arguments. With the `down2Q` sliver defined there, `inner-product` could be rewritten as:

```
(define (inner-product synq1 synq2)
  (down2Q 0
          (lambda (elt1 elt2 sum) (+ (* elt1 elt2) sum))
          synq1
          synq2))
```

Not all slivers exhibiting fan-in can be defined in terms of `map2Q`. Consider the `appendQ` sliver, whose behavior is exhibited in Figure 6.16. `AppendQ` only begins processing its second input synquence after completely processing the first. Note how the barrion label associated with the synquence generated by `to-1` (`A`) differs from the label associated with synquence generated by `splay-list` (`B`). Because the two synquences are not processed in lock step, the corresponding barrions are never unified. Because `map2Q` processes both inputs in lock step, it cannot be used to define `appendQ` or other slivers with special processing requirements for multiple synquence inputs.

```
OPERA> (down-+ (appendq (to-1 2) (splay-list '(7 5))))
-----------------------:down[A,0]
(zero? 2) --> ()
(-1+ 2) --> 1
(+ 2 0) --> 2
-----------------------:down[A,1]
(zero? 1) --> ()
(-1+ 1) --> 0
(+ 1 2) --> 3
-----------------------:down[A,2]
(zero? 0) --> #t
-----------------------:down[B,0]
(null? (7 5)) --> ()
(cdr (7 5)) --> (5)
(car (7 5)) --> 7
(+ 7 3) --> 10
-----------------------:down[B,1]
(null? (5)) --> ()
(cdr (5)) --> ()
(car (5)) --> 5
(+ 5 10) --> 15
-----------------------:down[B,2]
(null? ()) --> #t
; Value: 15
```

Figure 6.16: A sample trace of `appendQ`.

6.1.5 Fan-out

The lock step behavior and laziness illustrated in many of the examples up to this point is not very remarkable. In most cases, similar behavior could be obtained using standard lazy data structures (e.g., Scheme streams). The advantages of synquences over lazy lists are best illustrated by sliver programs that exhibit fan-out. In this case, fan-out means either that (1) a single synquence produced by one sliver is consumed by multiple slivers elsewhere in a sliver network or (2) a sliver produces more than one synquence.

A simple example of fan-out is a program that iteratively computes the average of a list of numbers (see the sliver diagram in Figure 5.18 on page 5.18). Each of the three slivers maintains one state variable of the iteration (current list, current sum, and current count). The diagram exhibits fan-out because the synquence output of `SPLAY-LIST` is use as an input to both `DOWN-SUM` and `DOWN-COUNT`.

A SYNAPSE program for the list averaging problem, along with a sample execution trace, appears in Figure 6.17. The trace, which corresonds to the snapshot movie in Figure 5.19, clearly shows that the network as a whole behaves like a monolithic iteration. In contrast, a version using lazy lists rather than synquences could build up space linear in the size of the input list. The synchronization information propagated by slags guarantees that all the slivers work in concert; one cannot race ahead or lag behind the others. In this respect, synquences resemble interprocess communication channels with bounded buffering. Waters [Wat91] and Hughes [Hug84] also extend the aggregate data approach with forms of synchronization in order to achieve this effect.

One advantage of slivers over the channel-based approaches and the other aggregate data extensions is the ease with which recursion is handled in networks exhibiting fan-out. (A more important advantage is the handling of tree-structured compuations, to be discussed in Section 6.2.) For example, the list-averaging example can be turned into a recursive process by changing one or both of the accumulating slivers to have *up* shape. Figure 6.18 contains an example that corresponds to the snapshot movie in Figure 5.20. The up barriers guarantee that the accumulations of the *up* phase proceed in lock step. This accurately models the behavior of a monolithic recursion that takes a list and returns

```
(define (average-list-iter lst)
  (let ((nums (splay-list lst)))
    (/ (down-+ nums)
       (down-length nums))))

; DOWN-+ as before

(define (down-length synq)
  (downQ 0
         (lambda (ignore count) (1+ count))
         synq))

OPERA> (average-list-iter '(7 2 6))
-----------------------:down[A,0]
(null? (7 2 6)) --> ()
(cdr (7 2 6)) --> (2 6)
(1+ 0) --> 1
(car (7 2 6)) --> 7
(+ 7 0) --> 7
-----------------------:down[A,1]
(null? (2 6)) --> ()
(cdr (2 6)) --> (6)
(car (2 6)) --> 2
(+ 2 7) --> 9
(1+ 1) --> 2
-----------------------:down[A,2]
(null? (6)) --> ()
(cdr (6)) --> ()
(car (6)) --> 6
(+ 6 9) --> 15
(1+ 2) --> 3
-----------------------:down[A,3]
(null? ()) --> #t
(/ 15 3) --> 5
; Value: 5
```

Figure 6.17: An iterative list averaging program and sample trace. The trace corresponds to the snapshot movie in Figure 5.19.

both its sum and its length.[10]

The lock step behavior of slivers extends to any shape-compatible network of slivers arranged in a directed acyclic graph (DAG). Figure 6.19 shows a complex network exhibiting fan-in, fan-out, and undirected cycles (i.e., slivers between which there are multiple paths). The corresponding trace in Figure 6.20 illustrates the iterative behavior of the network.

6.1.6 Deadlock

In all of the examples presented above, slivers networks behaved like monolithic recursions only because the slivers were carefully connected in shape-compatible ways. But not all arrangements of slivers are shape-compatible. Consider the following example, which is based on the arrangement shown earlier in Figure 5.10:

```
OPERA> (down-+ (mapQ square (up-scanQ 1 * (splay-list '(7 2 6)))))
----------------------:down[A,0]
(null? (7 2 6)) --> ()
(car (7 2 6)) --> 7
(cdr (7 2 6)) --> (2 6)
++++++++++++++++++++++++++++++++
DEADLOCK! -- pcall:38
++++++++++++++++++++++++++++++++
```

The `down-+` sliver requires synquence elements in the *down* phase of the computation, but the `upQ` sliver does not produce any elements until the *up* phase. The upshot is that computation reaches a deadlock state from which no progress can be made. In the above trace, this state is marked by the `DEADLOCK!` indicator. (The information following the indicator is debugging information that we will ignore.)

SYNAPSE employs a dynamic deadlock detection strategy. This means that a computation may be able to make some progress before deadlock is signalled. In the above example, the `splay-list` generator is able to perform some list operations before a deadlock state is reached. Dynamic deadlock detection also means that a program may deadlock for some inputs but not for others. If we make the input list empty in the above example, the resulting computation will terminate normally because the circular dependency is never discovered:

[10] Lock step processing in the *up* phase is not always desirable. It is possible to design slivers that synchronize in the *down* phase but not in the *up* phase, but I do not include examples of such slivers in this report.

```
(define (average-list-rec lst)
  (let ((nums (splay-list lst)))
    (/ (up-+ nums)
       (up-length nums))))

(define (up-+ nums)
  (upQ 0 + nums))

(define (up-length nums)
  (upQ 0
       (lambda (ignore count) (1+ count))
       synq))

OPERA> (average-list-rec '(7 2 6))
-----------------------:down[A,0]
(null? (7 2 6)) --> ()
(cdr (7 2 6)) --> (2 6)
(car (7 2 6)) --> 7
-----------------------:down[A,1]
(null? (2 6)) --> ()
(cdr (2 6)) --> (6)
(car (2 6)) --> 2
-----------------------:down[A,2]
(null? (6)) --> ()
(car (6)) --> 6
(cdr (6)) --> ()
-----------------------:down[A,3]
(null? ()) --> #t
-----------------------:up[A,3]
(1+ 0) --> 1
(+ 6 0) --> 6
-----------------------:up[A,2]
(+ 2 6) --> 8
(1+ 1) --> 2
-----------------------:up[A,1]
(+ 7 8) --> 15
(1+ 2) --> 3
-----------------------:up[A,0]
(/ 15 3) --> 5
; Value: 5
```

Figure 6.18: A recursive list averaging program and sample trace. The trace corresponds to the snapshot movie in figure 5.20.

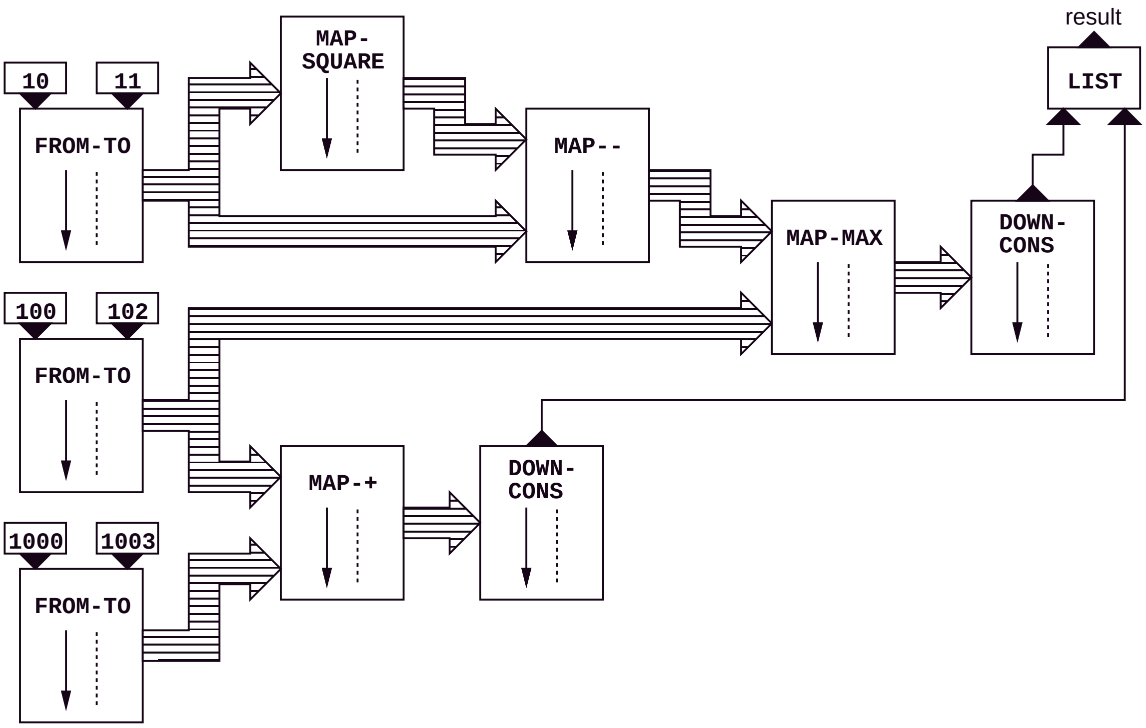


Figure 6.19: Complex sliver network exhibiting fan-in, fan-out, and undirected cycles. Since all the components are iterative, the network will generate an iterative computation.

```
OPERA> (let ((nums1 (from-to 10 11))
             (nums2 (from-to 100 102))
             (nums3 (from-to 1000 1003)))
         (list (down-cons (map2Q max
                                 (map2Q -
                                        (mapQ square nums1)
                                        nums1)
                                 nums2))
               (down-cons (map2Q + nums2 nums3))))
-----------------------:down[A,0]
(> 100 102) --> ()
(> 1000 1003) --> ()
(> 10 11) --> ()
(1+ 1000) --> 1001
(1+ 100) --> 101
(+ 100 1000) --> 1100
(tcons 1100 ()) --> (1100)
(1+ 10) --> 11
(square 10) --> 100
(- 100 10) --> 90
(max 90 100) --> 100
(tcons 100 ()) --> (100)
-----------------------:down[A,1]
(> 11 11) --> ()
(> 1001 1003) --> ()
(> 101 102) --> ()
(1+ 101) --> 102
(1+ 1001) --> 1002
(+ 101 1001) --> 1102
(tcons 1102 (1100)) --> (1102 1100)
(1+ 11) --> 12
(square 11) --> 121
(- 121 11) --> 110
(max 110 101) --> 110
(tcons 110 (100)) --> (110 100)
-----------------------:down[A,2]
(> 1002 1003) --> ()
(> 12 11) --> #t
(> 102 102) --> ()
(1+ 102) --> 103
(1+ 1002) --> 1003
(+ 102 1002) --> 1104
(tcons 1104 (1102 1100)) --> (1104 1102 1100)
-----------------------:down[A,3]
(> 1003 1003) --> ()
(> 103 102) --> #t
(cons (1104 1102 1100) ()) --> ((1104 1102 1100))
(cons (110 100) ((1104 1102 1100))) --> ((110 100) (1104 1102 1100))
; Value: ((110 100) (1104 1102 1100))
```

Figure 6.20: Sample trace generated by a computation specified by a complex sliver network.

```
OPERA> (down-+ (mapQ square (up-scanQ 1 * (splay-list '()))))
----------------------:down[A,0]
(null? ()) --> #t
----------------------:up[A,0]
; Value: 0
```

From a high-level perspective, the source of the deadlock in the first example above is that the computation defined by any sliver network is only allowed to buffer information in a single stack. In the *down* phase, the computation is certainly capable of buffering the outputs of `splay-list` and stacking the pending `*` operations. But the program additionally requires the *up* phase of the computation to buffer the outputs of the `up-scanQ` sliver and stack the pending `+` and `square` operations. Intuitively, this behavior requires *two* stacks, and cannot be expressed with the single stack that characterizes a monolithic recursion. The deadlock of this example is not a blemish, but is dictated by the goal of operational faithfulness.

Since the order of summation does not matter in the example, deadlock can easily be avoided by using `up-+` rather than `down-+` to accumulate the sum. More generally, however, deadlock cannot be fixed with so simple a change. Consider the following `down-up-scan-+` procedure:

```
(define (down-up-scan-+ lst)
  (down-cons! '()
              (down-scanQ 0 +
                          (up-scanQ 0 +
                                    (splay-list lst)))))
```

A correct result for this procedure requires that the order of the up and down summations are preserved. However, as written, the procedure will clearly deadlock on a non-empty input.

It is almost always possible to circumvent deadlocks in sliver networks by explicitly buffering intermediate data.[11] Here is the definition of a simple buffer that first collects all of the elements of a synquence into a list and then generates a new synquence from that list:

```
(define (list-buffer synq)
  (splay-list (up-cons synq)))
```

[11]Buffering techniques cannot circumvent deadlocks due to *unresolvable circularities* in which the input to a primitive operator fundamentally depends on its output.

Figure 6.21 shows how a `list-buffer` sliver can be used to avoid deadlock by breaking the single lock step component of `down-up-scan-+` into two such components. Each component to an independent lock step processing stage of the computation. The non-deadlocking version of `down-up-scan-+` and a sample trace appear in Figure 6.22.

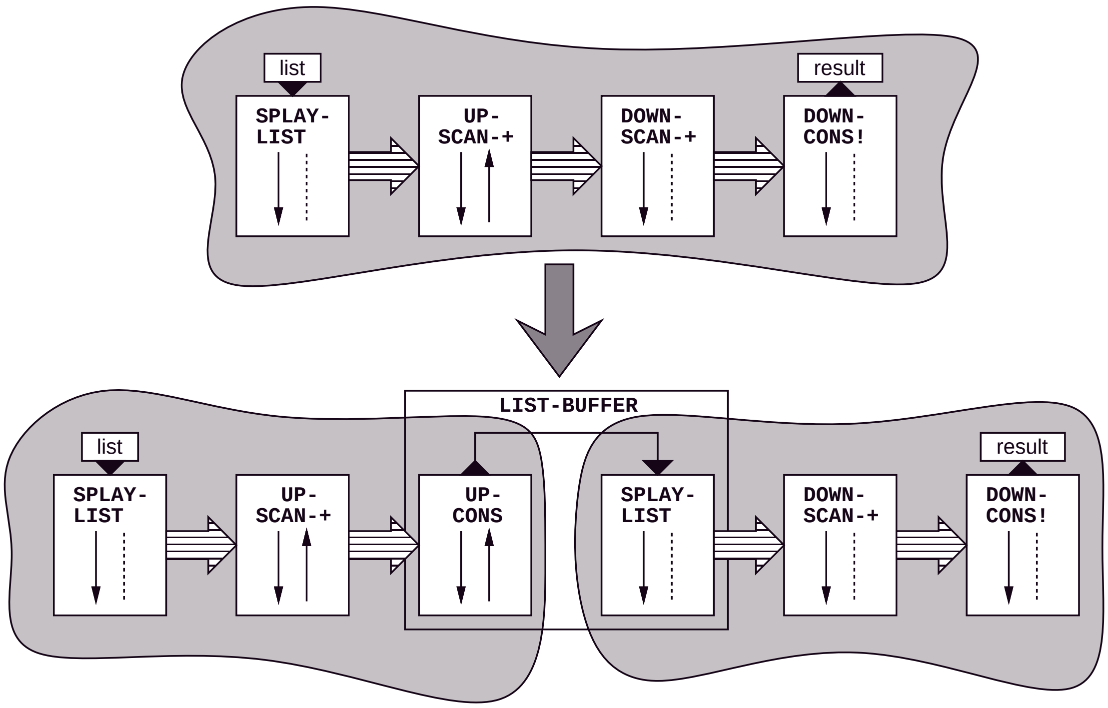


Figure 6.21: A `LIST-BUFFER` sliver can be used to split a single lock step component (shaded blob) into two lock step components.

Specifying explicit buffers may seem annoying, but it is the cost of the space consumption guarantees provided by sliver decomposition. The implicit buffering provided by other aggregate data approaches makes it hard to control the storage profiles of modular programs. Existing list and tree removal techniques (such as listlessness [Wad84, Wad85], deforestation [Wad88, Chi92, GLJ93], and other transformations [Bel86, Bir89a]) automatically remove some intermediate data structures, but they either are limited to restricted network topologies or do not guarantee that all intermediate structures go away. In con-

```
;;; A non-deadlocking version of the DOWN-UP-SCAN-+ presented in the text.
(define (down-up-scan-+ lst)
  (down-cons! '()
              (down-scanQ 0 +
                          (list-buffer
                           (up-scanQ 0 +
                                     (splay-list lst))))))

OPERA> (down-up-scan-+ '(1 20))
-----------------------:down[A,0]
(null? (1 20)) --> ()
(car (1 20)) --> 1
(cdr (1 20)) --> (20)
-----------------------:down[A,1]
(null? (20)) --> ()
(car (20)) --> 20
(cdr (20)) --> ()
-----------------------:down[A,2]
(null? ()) --> #t
-----------------------:up[A,2]
(+ 20 0) --> 20
(tcons 20 ()) --> (20)
-----------------------:up[A,1]
(+ 1 20) --> 21
(tcons 21 (20)) --> (21 20)
-----------------------:up[A,0]
(cons ignore ()) --> (ignore)
-----------------------:down[B,0]
(null? (21 20)) --> ()
(car (21 20)) --> 21
(+ 21 0) --> 21
(cdr (21 20)) --> (20)
(tcons 21 ()) --> (21)
(set-cdr! (ignore) (21)) --> #t
-----------------------:down[B,1]
(null? (20)) --> ()
(car (20)) --> 20
(+ 20 21) --> 41
(cdr (20)) --> ()
(tcons 41 ()) --> (41)
(set-cdr! (21) (41)) --> #t
-----------------------:down[B,2]
(null? ()) --> #t
(cdr (ignore 21 41)) --> (21 41)
; Value: (21 41)
```

Figure 6.22: Deadlock can often be avoided by using explicit buffering. Here `list-buffer` resolves the shape incompatibility between `up-scanQ` and `down-scanQ`.

trast, slivers provide a framework in which programmers manipulate shaped components to avoid or insert buffering. This is an approach similar to that embraced by Waters's series package, except that series only support linear, iterative computations whereas slivers support recursions and tree-shaped computations.

In linear computations, deadlock is typically induced by an attempt to use in the *down* phase a result computed in the *up* phase. But there are other dependencies that can lead to deadlock. For instance, since `appendQ` begins processing its second argument only after it completes processing of its first argument, a deadlock will be induced if the two arguments share synchronization information. The following expression deadlocks because the `to-1` generator can't both immediately produce elements to feed the first argument of `appendQ` but delay producing the same elements until the second argument is needed (see Figure 6.23):

```
(let ((nums (to-1 3)))
  (down-+ (appendQ nums nums)))
```

This deadlock can be removed by wrapping a `list-buffer` around the second argument.

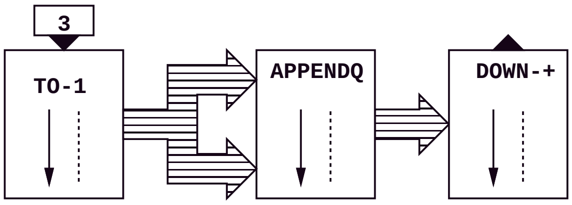
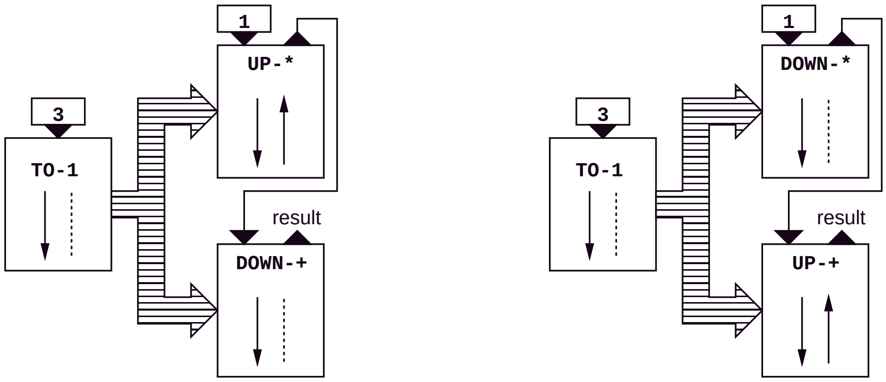


Figure 6.23: An example of deadlock involving `appendQ`.

Another source of deadlock is wiring the non-slag output of one reducer to the non-slag input of another sliver in the same network. Here are two expressions of this sort that give rise to deadlock (see Figure 6.24):

```
(let ((nums (to-1 3)))
  (downQ (upQ 1 *  nums) + nums))

(let ((nums (to-1 3)))
  (upQ (downQ 1 * nums) + nums))
```

Due to the strict nature of call boundaries, both `downQ` and `upQ` require values for all arguments before proceeding. But each example requires the final value of processing a slag to be availble before processing on the slag begins! This is a recipe for deadlock. Explicit buffering can be used to avoid deadlock in both cases.



(a) Non-slag dataflow from *up* to *down*. **(b)** Non-slag dataflow from *down* to *up*.

Figure 6.24: Two examples of deadlock caused by non-slag dataflow.

The deadlocking network in Figure 6.24(**b**) can be fixed by a non-buffering method. Since the initial value of the `upQ` isn't really required until the beginning of the *up* phase, there's no intrinsic reason why that value can't be supplied by a *down* computation (which is guaranteed to return before the *up* phase begins). The deadlock in this example is actually a spurious one caused by the strictness convention of call boundaries. In this case, it is safe to override the convention by introducing explicit non-strictness. Figure 6.25 shows how wrapping an `eagon` around the result of the `downQ` allows the computation to proceed without deadlock. `Eagon` is an OPERA special form that starts computing its argument expression but immediately returns a placeholder for its result.[12] While its value is being computed, the placeholder can be passed into and returned from procedures, and stored into and selected from data structures. Any operation that actually needs to examine the placeholder's value must wait for its computation to complete. Using eagon in the example "fools" the call boundary into "believing" that a value is available when it really isn't. But this is fine, since the placeholder's value isn't required until after the *down* phase is complete, by which time the placeholder value will be computed.

[12]Those familiar with *futures* will recognize that eagon is just another name for future. See the discussion of eagons in Chapter 7.

```
OPERA> (let ((nums (to-1 3)))
         (upQ (eagon (downQ 1 * nums)) + nums))
-----------------------:down[A,0]
(zero? 3) --> ()
(-1+ 3) --> 2
(* 3 1) --> 3
-----------------------:down[A,1]
(zero? 2) --> ()
(-1+ 2) --> 1
(* 2 3) --> 6
-----------------------:down[A,2]
(zero? 1) --> ()
(-1+ 1) --> 0
(* 1 6) --> 6
-----------------------:down[A,3]
(zero? 0) --> #t
-----------------------:up[A,3]
(+ 1 6) --> 7
-----------------------:up[A,2]
(+ 2 7) --> 9
-----------------------:up[A,1]
(+ 3 9) --> 12
-----------------------:up[A,0]
; Value: 12
```

Figure 6.25: OPERA's non-strict `eagon` form can be used to fix a spurious deadlock.

6.1.7 Filtering

Synquence slivers are designed to handle filtering in a reusable manner (see Section 5.5.3). Gaps are introduced into a synquence by the `filterQ` sliver and are propagated by all transducers except for scanners and `reifyQ`.

The following trace shows a simple example of a filtering:

```
OPERA> (downQ 0 + (mapQ square (filterQ even? (to-1 3))))
-----------------------:down[A,1]
(zero? 3) --> ()
(even? 3) --> ()
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(even? 2) --> #t
(square 2) --> 4
(+ 4 0) --> 4
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(even? 1) --> ()
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
```

The computation sums the squares of all the even numbers in a given synquence. As indicated by the trace, the mapper and summer ignore the slots associated with the odd inputs.

A scanner fills a gapped slot with the current value of its accumulator:

```
OPERA> (downQ 0 + (down-scanQ 1 max (mapQ square (filterQ even? (to-1 3)))))
-----------------------:down[A,1]
(zero? 3) --> ()
(even? 3) --> ()
(+ 1 0) --> 1
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(even? 2) --> #t
(square 2) --> 4
(max 4 1) --> 4
(+ 4 1) --> 5
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(even? 1) --> ()
(+ 4 5) --> 9
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
; Value: 9
```

Although the `max` operation of the scanner is only performed in the second slot, the output of the scanner is a synquence of the elements `1`, `4`, and `4`. These are summed by `downQ` to yield a `9`.

Sometimes it is desirable for a scanner to preserve the gaps of its inputs. This can be accomplished with the help of the handy `preserving-gaps` procedure:

```
(define (preserving-gaps scanner)
  (lambda (init op synq)
    (map2Q (lambda (a b) a)
           (scanner init op synq)
           synq)))
```

`Preserving-gaps` maps a given scanner into a gap-preserving scanner using the gap-preserving properties of the `map2Q` sliver (see Figure 5.22). If this procedure is used in the above example, the synquence consumed by the summer will contain a `4` wedged between two gaps:

```
OPERA> (downQ 0 +
              ((preserving-gaps down-scanQ)
               1
               max
               (mapQ square (filterQ even? (to-1 3)))))
-----------------------:down[A,1]
(zero? 3) --> ()
(even? 3) --> ()
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(even? 2) --> #t
(square 2) --> 4
(max 4 1) --> 4
(+ 4 0) --> 4
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(even? 1) --> ()
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
; Value: 4
```

Up accumulators and scanners exhibit the desirable behavior of aggresively testing for gaps in the down direction. In the following trace, there is no pending stack frame corresponding to the the 2 input because a `upQ` effectively makes a tail call for that input:

```
OPERA> (upQ 0 + (filterQ odd? (to-1 3)))
-----------------------:down[A,1]
(zero? 3) --> ()
(odd? 3) --> #t
(-1+ 3) --> 2
-----------------------:down[A,2]
(zero? 2) --> ()
(odd? 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,3]
(zero? 1) --> ()
(odd? 1) --> #t
(-1+ 1) --> 0
-----------------------:down[A,4]
(zero? 0) --> #t
-----------------------:up[A,4]
(+ 1 0) --> 1
-----------------------:up[A,2]  ; *** Missing frame here
(+ 3 1) --> 4
-----------------------:up[A,1]
; Value: 4
```

This desirable behavior is even exhibited by parallel combinations of slivers that independently filter a shared input. For example, the computation in Figure 6.26 is a parallel combination of a filtered up sum accumulator and a filtered up product accumulator. Note a stack frame is pushed in the resulting computation only if one of the accumulators has a pending operation to perform in that frame. No frame is pushed for slots in which both accumulators process a gap (e.g. the slot for which `to-1` produces 2).

```
OPERA> (let ((nums (to-1 4)))
         (cons (upQ 0 + (filterQ odd? nums))
               (upQ 1 * (filterQ (lambda (x) (> x 2)) nums))))
-----------------------:down[A,1]
(zero? 4) --> ()
(> 4 2) --> #t
(odd? 4) --> ()
(-1+ 4) --> 3
-----------------------:down[A,2]
(zero? 3) --> ()
(> 3 2) --> #t
(odd? 3) --> #t
(-1+ 3) --> 2
-----------------------:down[A,3]
(zero? 2) --> ()
(odd? 2) --> ()
(> 2 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,4]
(zero? 1) --> ()
(odd? 1) --> #t
(> 1 2) --> ()
(-1+ 1) --> 0
-----------------------:down[A,5]
(zero? 0) --> #t
-----------------------:up[A,5]
(+ 1 0) --> 1
-----------------------:up[A,3]  ; *** Missing frame here
(+ 3 1) --> 4
(* 3 1) --> 3
-----------------------:up[A,2]
(* 4 3) --> 12
-----------------------:up[A,1]
(cons 4 12) --> (4 . 12)
; Value: (4 . 12)
```

Figure 6.26: Trace illustrating the proper interleaving of independently filtered parallel up computations. Note that there is no up frame for the slot where both filter tests fail.

Despite the apparent aggressive testing in the *down* phase, `upQ` and `up-scanQ` still deal appropriately with gaps that are determined in the *up* phase. Figure 6.27 shows a trace of a computation with filtering in both the *down* and *up* phases (the computation is similar to that of the sliver diagram depicted in Figure 5.26). The `odd?` tests that occur in the *down* phase determined that there are only three stacked frames. The `even?` tests are required to occur in the *up* phase. Only one `*` operation is performed because only one output of the scanner is even.

Achieving these kinds of expected fine-grained operational behavior for filtered synquences is one of the technical triumphs of SYNAPSE. See Section 7.2.4 for an explanation of how it all works.

6.2 Tree Computations

SYNAPSE supports tree computations through a suite of slivers that manipulate syndrites (synchronized lazy trees). These are summarized in Figures 6.28 – 6.30.

Syndrite slivers are natural extensions to the synquence slivers presented in the previous section. Syndrite generators, mappers, and filters are just tree-shaped versions of the corresponding synquence operators. However, syndrite accumulators and scanners support a much richer shape vocabulary than their linear cousins. For example, tree nodes can be processed in parallel either down or up the branches of a tree; or they can be processed in a wide variety of sequential traversals.

We will only consider sequential traversals that follow the same node-processing strategy at each node. Such traversals can be characterized by two properties:

1. The *order* of a sequential traversal specifies when the element of a node is processed relative to the processing of its children. There are three classes of orders:

   (a) *pre-order*: an element of a node is processed before any of its children's elements.

   (b) *in-order*: an element of a node is processed after some of its children's elements but before other of its children's elements.

   (c) *post-order*: an element of a node is processed after all of its children's elements.

```
OPERA> (upQ 1 * (filterQ even? ((preserving-gaps up-scanQ)
                                0
                                +
                                (filterQ odd? (to-1 5)))))
-----------------------:down[A,1]
(zero? 5) --> ()
(odd? 5) --> #t
(-1+ 5) --> 4
-----------------------:down[A,2]
(zero? 4) --> ()
(odd? 4) --> ()
(-1+ 4) --> 3
-----------------------:down[A,3]
(zero? 3) --> ()
(odd? 3) --> #t
(-1+ 3) --> 2
-----------------------:down[A,4]
(zero? 2) --> ()
(odd? 2) --> ()
(-1+ 2) --> 1
-----------------------:down[A,5]
(zero? 1) --> ()
(odd? 1) --> #t
(-1+ 1) --> 0
-----------------------:down[A,6]
(zero? 0) --> #t
-----------------------:up[A,6]
(+ 1 0) --> 1
(even? 1) --> ()
-----------------------:up[A,4]
(+ 3 1) --> 4
(even? 4) --> #t
(* 4 1) --> 4
-----------------------:up[A,2]
(+ 5 4) --> 9
(even? 9) --> ()
-----------------------:up[A,1]
; Value: 4
```

Figure 6.27: Trace illustrating the desirable properties of synquence filtering.

Generators:

(`genDi` *initial-state element&next-states*) [*parallel down*]
Generates a syndrite whose element values are computed from a "state" that is passed down the tree in a *parallel down* fashion. At every syndrite node, the node element and child states are determined by the *element&next-states* procedure, which takes the current state of the node and a two-argument receiver, and calls the receiver on the desired element and a list of child states. A syndrite branch terminates when the list of child states is empty.

(`genDe` *initial-state element&next-states done?*) [*parallel down*]
Like `genDi`, except applies *done?* to the node state before calling *element&next-states*. If *done?* returns true, then the current branch is terminated with a gap element; otherwise *element&next-states* is used to compute the element and next states.

Figure 6.28: A summary of SYNAPSE's syndrite generating slivers.

The *pre* and *post* sliver shapes indicate the pre-order and post-order strategies, respectively. The *in* shape of syndrite slivers corresponds to processing a node element after the processing the first child but before processing the rest of the children. This is only one instance of the in-order strategy, but it has the nice property that for binary syndrites it corresponds to the traditional notion of in-order processing on binary trees [Knu73].

2. The *direction* of a sequential traversal specifies the order in which a node's children are processed. If the children are assumed to have a default left-to-right ordering, then the direction can be specified as a permutation on this ordering. Any permutation is allowed, but in our examples, we will only consider the identity permuation (a *left-to-right* traversal) and a reversing permutation (a *right-to-left* traversal).

In SYNAPSE, the *direction* argument to various sequential transducers and reducers is expected to be a *pair* of procedures: a list permutation and its corresponding inverse permutation. The inverse permutation is used by transducers to reconstruct the original ordering of children after permuting them. Here's how the two standard directions are defined:

Reducers:

(`upD` *op gap-op synd*) [*parallel up*]
Recursively accumulates the elements of *synd* using *op* and *gap-op*. At every node whose element is not a gap, *op* is applied to the node element and a list of the accumulated results for each child. At a node whose element is a gap, *gap-op* is applied to a list of the accumulated results for each child.

(`preD` *direction init op synd*) [*direction pre*]
Starting with *init*, uses *op* to accumulate the elements of *synd* in a pre-order walk over the syndrite. A pre-order walk processes the element at a node before processing and children. At every node, *op* is applied to the element and the current accumulated value. *Direction* specifies the direction of the walk (e.g., left-to-right, right-to-left); see the text to see how directions are specified.

(`inD` *direction init op synd*) [*direction in*]
An in-order sequential accumulation over *synd*. An in-order walk processes the element at a node after processing the first child but before processing the other children.

(`postD` *direction init op synd*) [*direction post*]
A post-order sequential accumulation over *synd*. A post-order walk processes the element at a node after processing all of the children.

(`up-firstD` *synd*) [*parallel up*]
Returns the top element of a syndrite. Signals an error if the top element is a gap.

(`pre-lastD` *direction synd*) [*direction pre*]
Returns the last (non-gap) element processed in a *direction* pre-order walk over *synd*. Signals an error if no element is found.

(`in-lastD` *direction synd*) [*direction in*]
In-order version of `pre-lastD`.

(`post-lastD` *direction synd*) [*direction post*]
Post-order version of `pre-lastD`.

(`pre-nthD` *direction index synd*) [*direction pre*]
Returns the *index*th (non-gap) element processed in a *direction* pre-order walk over *synd*. Signals an error if *synd* does not contain *index* elements.

(`in-nthD` *direction index synd*) [*direction in*]
In-order version of `pre-nthD`.

(`post-nthD` *direction index synd*) [*direction post*]
Post-order version of `pre-nthD`.

Figure 6.29: A summary of SYNAPSE's syndrite reducing slivers.

Transducers:

(`mapD` *fun synd*) [*across*]
Returns the syndrite resulting from the elementwise application of *fun* to *synd*.

(`map2D` *fun synd1 synd2*) [*across*]
Returns the syndrite resulting from the elementwise application of *fun* to corresponding elements of *synd1* and *synd2*. An output element is a gap if either input element is a gap.

(`filterD` *pred synd*) [*across*]
Returns a syndrite with the same structure as *synd* in which every element satisfying *pred* is mapped to itself and all other elements are mapped to gaps.

(`reifyD` *obj synd*) [*across*]
Returns a syndrite that maps every gap of *synd* to *obj* and every ungapped element to itself.

(`unreifyD` *obj synd*) [*across*]
Returns a syndrite that maps every instance of *obj* in *synd* to a gap and every other element to itself.

(`down-scanD` *init op synd*) [*parallel down*]
Returns the syndrite of intermediate accumulated values in an iterative accumulation of each branch of *synd* using combiner *op* and initial value *init*.

(`up-scanD` *op gap-op synd*) [*parallel up*]
Returns the syndrite of intermediate accumulated values in a parallel recursive accumulation of *synd* using the combiner *op* at ungapped nodes and the combiner *gap-op* at gapped nodes.

(`pre-scanD` *direction init op synd*) [*direction pre*]
Returns the syndrite of intermediate accumulated values in a sequential *direction* pre-order accumulation of *synd* using combiner *op* and initial value *init*.

(`in-scanD` *direction init op synd*) [*direction in*]
In-order version of `pre-scanD`.

(`post-scanD` *direction init op synd*) [*direction post*]
Post-order version of `pre-scanD`.

(`down-shiftD` *init synd*) [*parallel down*]
Returns a syndrite with the same structure at *synd*, but in which every ungapped element location has the value of the first ungapped element found on the path from the parent of the location to the root. If there is no ungapped element on the path to the root, *init* is used.

Figure 6.30: A summary of SYNAPSE's syndrite transducing slivers.

```
;;; The direction abstraction
(define (make-direction permute unpermute)
  (cons permuter unpermuter))

(define permuter car)
(define unpermuter car)

;;; Left-to-right direction
(define l->r
  (make-direction (lambda (lst) lst) (lambda (lst) lst)))

;;; Right-to-left direction
(define r->l
  (make-direction reverse reverse))
```

6.2.1 Simple Examples

Before investigating applications of the syndrite slivers, we first exercise them in some simple situations. The examples will employ the following routines, which represent one way to convert between s-expressions (parenthesized tree notation) and syndrites:

```
(define (splay-tree top-sexp)
  (genDi top-sexp
         (lambda (sexp return)
           (if (pair? sexp)
               (return (car sexp) (cdr sexp))
               (return sexp '())))))

(define (glom-tree synd)
  (upD (lambda (elt subtrees)
         (if (null? subtrees)
             elt
             (cons elt subtrees)))
       (lambda (subtrees)
         (if (null? subtrees)
             '*gap*
             (cons '*gap* subtrees)))
       synd))
```

Let A stand for an atom (non-pair) and S_i stand for any s-expression. Then `splay-tree` treats A as a syndrite leaf node whose element is A and (S_0 S_1 $\ldots S_n$) as an syndrite intermediate node whose element is S_0 and whose children are the recursive results of converting S_1 $\ldots$ S_n. The element structure of the syndrite resulting from applying `splay-tree` to `(a (b c d) e (f (g h)))` is shown in Figure 6.31. `Glom-tree` performs the inverse conversion, but fills any gapped positions with the symbol `*gap*`.

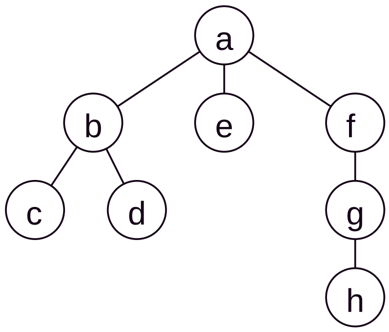


Figure 6.31: The tree corresponding to the syndrite created by `(splay-tree '(a (b c d) e (f (g h))))`.

`Splay-tree` gives one example of the use of a syndrite generator. Here are two more examples, which underscore the difference between `genDi` and `genDe`:

```
(glom-tree (genDi 1
                  (lambda (n return)
                    (if (> n 4)
                        (return n '())
                        (return n (list (* 2 n) (+ 1 (* 2 n))))))))
; Value: (1 (2 (4 8 9) 5) (3 6 7))

(glom-tree (genDe 1
                  (lambda (n return)
                    (return n (list (* 2 n) (+ 1 (* 2 n)))))
                  (lambda (n) (> n 4))))
; Value: (1 (2 (4 *gap* *gap*) *gap*) (3 *gap* *gap*))
```

Both examples produce a breadth index tree (see page 155), in which the number at each node is doubled before being passed to the left subtree, and is doubled and incremented before being passed to the right subtree. The `genDi` version terminates the tree when a number larger than 4 is encountered, but includes the number in the tree. The `genDe` version excludes numbers larger than 4 by putting gaps in their positions.

Figure 6.32 illustrates the different syndrite accumulation strategies by performing `cons` accumulations over the syndrite created by `(splay-tree '(a b c))`. A `cons` accumulation with `upD` is similar to `glom-tree` except that the leaves of the resulting tree are parenthesized. (Also, the `no-gaps` gap accumulator is used to indicate that no gaps are expected in the input.) Left-to-right and right-to-left versions of `preD`, `inD`, and `postD` give all six possible permutations of a three element list. In the sequential accumulation examples, the numbers are listed in the *reverse* order that they are processed because later elements are prepended to the list of elements processed earlier. By using the `cdr`-bashing technique, it is possible for the accumulators to construct a list of the elements in the order visited.

```
OPERA> (define (sample-syndrite) (splay-tree '(a b c)))

OPERA> (define (no-gaps lst)
         (error "NO-GAPS: found a gap where none expected."))

OPERA> (upD cons no-gaps (abc))
; Value: (a (b) (c))

OPERA> (preD l->r '() cons (abc))
; Value: (c b a)

OPERA> (preD r->l '() cons (abc))
; Value: (b c a)

OPERA> (inD l->r '() cons (abc))
; Value: (c a b)

OPERA> (inD r->l '() cons (abc))
; Value: (b a c)

OPERA> (postD l->r '() cons (abc))
; Value: (a c b)

OPERA> (postD r->l '() cons (abc))
; Value: (a b c)
```

Figure 6.32: Examples of different shapes of syndrite accumulation.

Examples of syndrite scanning appear in Figure 6.33. In each case, the result is a tree of intermediate `cons` accumulations. The length of the list in each tree position indicates the order in which the nodes are visited. For example, the result `((a c b) (b) (c b))` indicates that the `b` node was visited first, the `c` node was visited second, and the `a` node was visited last.

Syndrite mapping and filtering operations are *across* operators that simply perform elementwise application:

```
OPERA> (glom-tree (down-scanD '() cons (abc)))
; Value: ((a) (b a) (c a))

OPERA> (glom-tree (up-scanD cons no-gaps (abc)))
; Value: ((a (b) (c)) (b) (c))

OPERA> (glom-tree (pre-scanD l->r '() cons (abc)))
; Value: ((a) (b a) (c b a))

OPERA> (glom-tree (pre-scanD r->l '() cons (abc)))
; Value: ((a) (b c a) (c a))

OPERA> (glom-tree (in-scanD l->r '() cons (abc)))
; Value: ((a b) (b) (c a b))

OPERA> (glom-tree (in-scanD r->l '() cons (abc)))
; Value: ((a c) (b a c) (c))

OPERA> (glom-tree (post-scanD l->r '() cons (abc)))
; Value: ((a c b) (b) (c b))

OPERA> (glom-tree (post-scanD r->l '() cons (abc)))
; Value: ((a b c) (b c) (c))
```

Figure 6.33: Examples of different shapes of syndrite scanning.

```
(define (sample) (splay-tree '(1 (2 3 4) 5 (6 (7 8)))))

OPERA> (glom-tree (mapD square (sample)))
; Value: (1 (4 9 16) 25 (36 (49 64)))

OPERA> (glom-tree (filterD even? (sample)))
; Value: (*gap* (2 *gap* 4) *gap* (6 (*gap* 8)))
```

In an ungapped tree, syndrite shifting moves each node element one level down every branch from the node. In the presence of gaps, an element replaces the next element accessible down each branch:

```
OPERA> (glom-tree (down-shiftD 17 (big-synd)))
; Value: (17 (1 2 2) 1 (1 (6 7)))

OPERA> (glom-tree (down-shiftD 17 (filterD even? (big-synd))))
; Value: (*gap* (17 *gap* 2) *gap* (17 (*gap* 6)))
```

6.2.2 Shape Combinations

Perhaps the most important feature of syndrite slivers is that their process shapes combine in a reasonable way. We will present a series concrete examples of traces to help build intuitions about these combinations.

We consider operations on a simple binary tree:

```
(define (binary) (splay-tree '(1 (2 3 4) (5 6 7))))
```

Figure 6.34 shows sample traces of left-to-right pre-order and post-order accumulations on `binary`.[13] Because of the way the tree is structured, the barrion index labelling a dotted barrier corresponds to the numeric element held by the syndrite with that barrion. For the pre-order case, the barriers indicate that accumulation of element n always happens immediately after the down rendezvous at the barrier labelled n. In the post-order case, the accumulation of element n occurs directly before the up rendezvous at the barrier labelled n.

The barriers also indicate that the pre-order accumulator makes some tail calls while the post-order accumulator does not. In the post-order case, every down barrier has a matching up barrier, signifying a post-order tree walker must return from every call. However, in

[13]The traces only show instances of the `+` and `*` operators.

```
OPERA> (preD l->r 1 * (binary))
-----------------------:down[A,1]
(* 1 1) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
-----------------------:up[A,3]
-----------------------:down[A,4]
(* 4 6) --> 24
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
-----------------------:down[A,6]
(* 6 120) --> 720
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
; Value: 5040

OPERA> (postD l->r 0 + (binary))
-----------------------:down[A,1]
-----------------------:down[A,2]
-----------------------:down[A,3]
(+ 3 0) --> 3
-----------------------:up[A,3]
-----------------------:down[A,4]
(+ 4 3) --> 7
-----------------------:up[A,4]
(+ 2 7) --> 9
-----------------------:up[A,2]
-----------------------:down[A,5]
-----------------------:down[A,6]
(+ 6 9) --> 15
-----------------------:up[A,6]
-----------------------:down[A,7]
(+ 7 15) --> 22
-----------------------:up[A,7]
(+ 5 22) --> 27
-----------------------:up[A,5]
(+ 1 27) --> 28
-----------------------:up[A,1]
; Value: 28
```

Figure 6.34: Traces of left-to-right pre-order and post-order accumulations of a binary tree.

the pre-order case, there are only three up barriers for the seven down barriers. These up barriers correspond to cases where the tree walker must return from a call, while the missing ones indicate calls for which the tree-walker does not return. In particular, the pre-order accumulator returns to top-level immediately after processing 7 because there is no work left to be done. In constrast, the post-order accumulator still needs to pop a stack of two pending accumulations after it processes 7.

Combining the above two processes in parallel retains the character of the individual processes. Figure 6.35 shows the trace of a sample parallel combination. Every operation still appears at the same relative location. The combined process returns from every call because one of the components does. If we instead combine two pre-order accumulators, the result is guaranteed to look like a single pre-order process (Figure 6.36). If we change the directions of the two accumulators to be incompatible, a deadlock results:

```
OPERA> (let ((bin (binary)))
         (cons (preD l->r 1 * bin)
               (postD r->l 0 + bin)))
-----------------------:down[A,1]
(* 1 1) --> 1
++++++++++++++++++++++++++++++++
DEADLOCK! -- prim-cons:52
++++++++++++++++++++++++++++++++
```

Using scanners, different tree walking processes can be combined in series. Figure 6.37 shows the trace of a series combination of a pre-order scanner and a post-order accumulator. The trace inherits its post-order shape from the accumulator. In contrast, a pre-order shape results from the series combination of a pre-order scanner and a pre-order accumulator (Figure 6.38). As expected, incompatible series combinations will lead to deadlock. For example, a pre-order accumulator would require the result of a post-order scanner sooner than the scanner could produce it:

```
OPERA> (preD l->r 1 *
             (post-scanD l->r 0 +
                         (binary)))
-----------------------:down[A,1]
++++++++++++++++++++++++++++++++
DEADLOCK! -- pcall:54
++++++++++++++++++++++++++++++++
```

```
OPERA> (let ((bin (binary)))
         (cons (preD l->r 1 * bin)
               (postD l->r 0 + bin)))


-----------------------:down[A,1]
(* 1 1) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 3 0) --> 3
-----------------------:up[A,3]
-----------------------:down[A,4]
(+ 4 3) --> 7
(* 4 6) --> 24
-----------------------:up[A,4]
(+ 2 7) --> 9
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
-----------------------:down[A,6]
(+ 6 9) --> 15
(* 6 120) --> 720
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
(+ 7 15) --> 22
-----------------------:up[A,7]
(+ 5 22) --> 27
-----------------------:up[A,5]
(+ 1 27) --> 28
-----------------------:up[A,1]
; Value: (5040 . 28)
```

Figure 6.35: Trace of a parallel combination of pre-order and post-order accumulators.

```
OPERA> (let ((bin (binary)))
         (cons (preD l->r 1 * bin)
               (preD l->r 0 + bin)))
-----------------------:down[A,1]
(+ 1 0) --> 1
(* 1 1) --> 1
-----------------------:down[A,2]
(+ 2 1) --> 3
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 3 3) --> 6
-----------------------:up[A,3]
-----------------------:down[A,4]
(+ 4 6) --> 10
(* 4 6) --> 24
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
(+ 5 10) --> 15
-----------------------:down[A,6]
(* 6 120) --> 720
(+ 6 15) --> 21
-----------------------:up[A,6]
-----------------------:down[A,7]
(+ 7 21) --> 28
(* 7 720) --> 5040
; Value: (5040 . 28)
```

Figure 6.36: Trace of a parallel combination of two pre-order accumulators.

```
OPERA> (postD l->r 0 +
              (pre-scanD l->r 1 *
                         (binary)))
-----------------------:down[A,1]
(* 1 1) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 6 0) --> 6
-----------------------:up[A,3]
-----------------------:down[A,4]
(* 4 6) --> 24
(+ 24 6) --> 30
-----------------------:up[A,4]
(+ 2 30) --> 32
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
-----------------------:down[A,6]
(* 6 120) --> 720
(+ 720 32) --> 752
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
(+ 5040 752) --> 5792
-----------------------:up[A,7]
(+ 120 5792) --> 5912
-----------------------:up[A,5]
(+ 1 5912) --> 5913
-----------------------:up[A,1]
; Value: 5913
```

Figure 6.37: Trace of a series combination of a pre-order scanner and a post-order accumulator.

```
OPERA> (preD l->r 0 +
             (pre-scanD l->r 1 *
                        (binary)))
-----------------------:down[A,1]
(* 1 1) --> 1
(+ 1 0) --> 1
-----------------------:down[A,2]
(* 2 1) --> 2
(+ 2 1) --> 3
-----------------------:down[A,3]
(* 3 2) --> 6
(+ 6 3) --> 9
-----------------------:up[A,3]
-----------------------:down[A,4]
(* 4 6) --> 24
(+ 24 9) --> 33
-----------------------:up[A,2]
-----------------------:down[A,5]
(* 5 24) --> 120
(+ 120 33) --> 153
-----------------------:down[A,6]
(* 6 120) --> 720
(+ 720 153) --> 873
-----------------------:up[A,6]
-----------------------:down[A,7]
(* 7 720) --> 5040
(+ 5040 873) --> 5913
; Value: 5913
```

Figure 6.38: Trace of a series combination of a pre-order scanner and a pre-order accumulator.

Mappers, filters, up accumulators, and parallel down and up scanners have malleable shapes that depend on the context in which they are used. For instance, Figure 6.39 shows how a down scanner conforms to shape constraints supplied by other slivers. When used in conjunction with a left-to-right accumulator, `down-scanD` is guaranteed to perform accumulations in a left-to-right order. However, with a right-to-left accumulator, `down-scanD` will perform its accumulations in a right-to-left order. If combined with a parallel up accumulator, the operations of `down-scanD` can happen in any order consistent with the data dependencies.

The final trace in Figure 6.39 illustrates an important fact about parallel tree shapes. If a lock step component contains only slivers with parallel down and up shapes, then the entire component has a parallel shape. This means that independent branches can be processed concurrently. For example, the sample trace shows that the processing of the `2` and `3` elements is interleaved. While such parallel processing can lead to more efficient execution times in the presence of multiple processors, it can also lead to much greater storage consumption. In the uniprocessor environment we have been assuming, the time efficiency vanishes but the space inefficiency remains. A programmer wishing to control space behavior must explicity add sequentiality to a parallel-shaped tree process in order to obtain desirable space behavior.

6.2.3 Extended Example: Alpha Renaming

As a practical example of syndrites, we show how the monolithic alpha renaming program introduced in Section 2.2 can be expressed as a modular SYNAPSE program. We also consider some extensions that are easily supported by the modular program.

A Sliver-based Alpha Renamer

Figure 6.40 is a sliver diagram for alpha renaming. `TERM->SYNDRITE` and `SYNDRITE->TERM` convert terms represented as s-expressions into syndrites. `FILTER-FORMALS` is a filter that passes only the formal parameters of abstraction nodes. `UNIQUE-NAMES` returns a tree of names such that the ungapped positions of its input map to unique outputs. Given a tree of names and a tree of values, `BIND` returns a tree of environments, where each environment

```
OPERA> (define (tiny) (splay-tree '(1 2 3))) ; Value: tiny

OPERA> (preD l->r 1 * (down-scanD 4 + (tiny)))
-----------------------:down[A,1]
(+ 1 4) --> 5
(* 5 1) --> 5
-----------------------:down[A,2]
(+ 2 5) --> 7
(* 7 5) --> 35
-----------------------:up[A,2]
-----------------------:down[A,3]
(+ 3 5) --> 8
(* 8 35) --> 280
; Value: 280

OPERA> (preD r->l 1 * (down-scanD 4 + (tiny)))
-----------------------:down[A,1]
(+ 1 4) --> 5
(* 5 1) --> 5
-----------------------:down[A,2]
(+ 3 5) --> 8
(* 8 5) --> 40
-----------------------:up[A,2]
-----------------------:down[A,3]
(+ 2 5) --> 7
(* 7 40) --> 280
; Value: 280

OPERA> (upD (lambda (elt lst) (* elt (*-list lst)))
            no-gaps
            (down-scanD 4 + (tiny)))
-----------------------:down[A,1]
(+ 1 4) --> 5
-----------------------:down[A,2]
-----------------------:down[A,3]
(+ 2 5) --> 7
(* 7 1) --> 7
(+ 3 5) --> 8
-----------------------:up[A,2]
(* 8 1) --> 8
-----------------------:up[A,3]
(* 8 1) --> 8
(* 7 8) --> 56
(* 5 56) --> 280
-----------------------:up[A,1]
; Value: 280
```

Figure 6.39: Traces illustrating the malleable nature of `down-scan`.

is a set of name/value bindings. At every position, an environment extends the bindings of its parent with the name and value at that position. `RENAME` takes a syntax tree and a tree of environments and returns a similarly shaped syntax tree in which each occurrence of a name (formal parameter and variable reference) is replaced by the corresponding value in the environment at the same position.

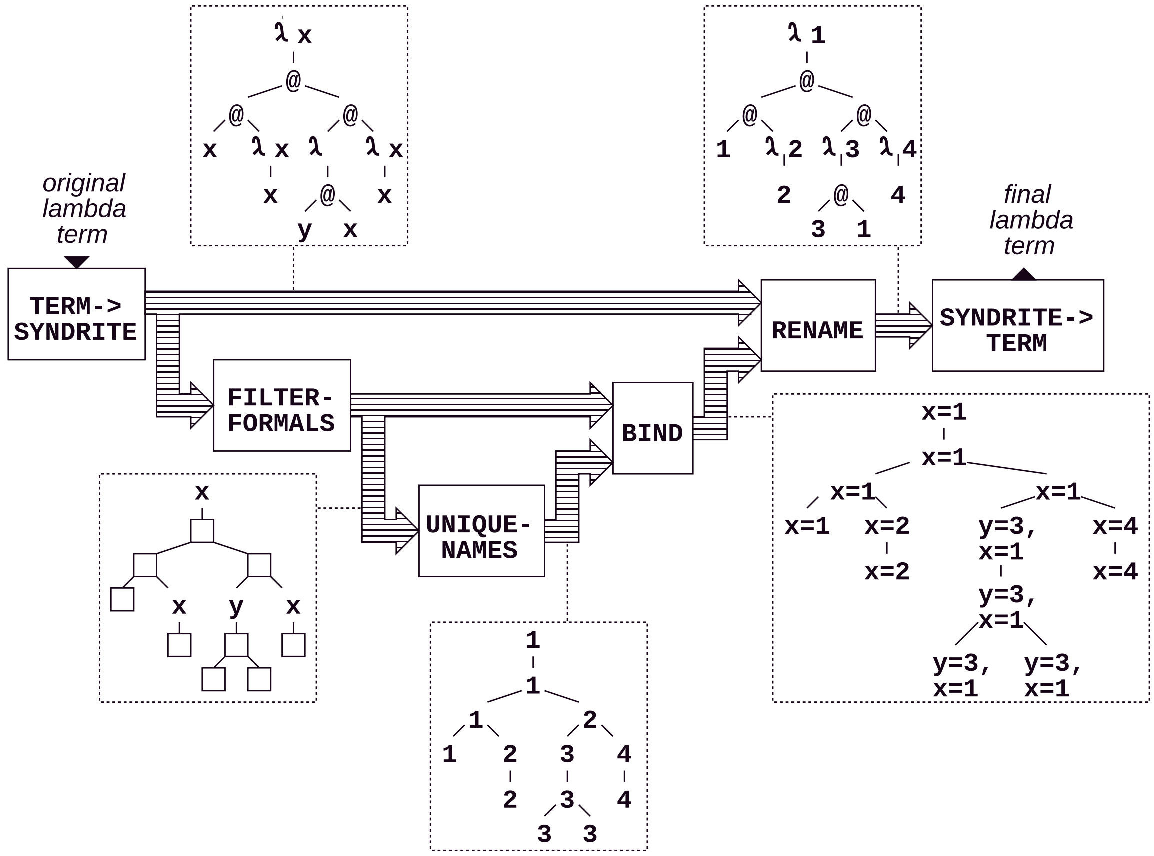


Figure 6.40: A signal processing style view of alpha renaming.

In the figure, each of the slags is annotated with a tree that represents the syndrite transmitted by the slag for sample input:

```
(lambda x
  (call (call x
              (lambda x x))
        (call (lambda y (call y x))
              (lambda x x)))))
```

The empty boxes in the output of FILTER-FORMALS indicate gaps. The names produced by UNIQUE-NAMES in this case are numbers that result from a left-to-right pre-order scan of an incrementer over the tree of formals.

The structure of the sliver diagram translates directly into a SYNAPSE program:

```
(define (alpha-rename term)
  (let ((nodes (term->syndrite term)))
    (let ((formals (filter-formals lambdrite)))
      (syndrite->term
       (rename nodes
               (bind formals
                     (unique-names formals)))))))
```

Each of the slivers used by alpha-rename can be easily constructed out of the SYNAPSE primitives. Term->syndrite transforms a term represented as an s-expression into a syndrite of nodes tagged with the type and contents of the corresponding position in the abstract syntax tree of the term:

```
(define (term->syndrite term)
  (genDi term
         (lambda (tm return)
           (cond
            ((variable? tm)
             (return (var-node (name tm))
                     '()))
            ((abstraction? tm)
             (return (lambda-node (formal tm))
                     (list (body tm))))
            ((application? tm)
             (return (call-node)
                     (list (rator tm) (rand tm))))))))
```

Name, formal, rator, etc. are selectors on terms, while var-node, lambda-node, and call-node are constructors for the syndrite elements. The inverse transformation is performed by syndrite->term:

```
(define (syndrite->term nodes)
  (upD
   (lambda (node subaccs)
     (cond
      ((var-node? node)
       (variable (node-name node)))
      ((lambda-node? node)
       (abstraction (node-formal node) (first subaccs)))
      ((call-node? node)
       (application (first subaccs) (second subaccs)))))
   no-gaps
   nodes))

(define (no-gaps lst) (error "Shouldn't happen"))
```

Here, `node-name` and `node-formal` are selectors on the syndrite nodes, while `variable`, `abstraction`, and `application` are term constructors.

Converting between terms and syndrites is the messy part; the slivers at the heart of the alpha renamer are remarkably elegant. `Filter-formals` is just a simple application of filtering and mapping:

```
(define (filter-formals nodes)
  (mapD node-formal
        (filterD lambda-node? nodes)))
```

`Unique-names` is easily implemented as a pre-order scanner that increments the accumulator every time it encounters a name in its input tree:

```
(define (unique-names names)
  (pre-scanD left-to-right
             0
             (lambda (x n) (1+ n))
             names))
```

`Bind` is a trivial application of a two-argument down scanner:

```
(define (bind names vals)
  (down-scan2D env-empty
               env-bind
               names
               vals))

(define (down-scan2D init combine synd1 synd2)
  (down-scanD init
              (lambda (pair acc)
                (combine (car pair) (cdr pair) acc))
              (map2D cons synd1 synd2)))
```

Here, `env-empty` is an empty environment and `env-bind` is a procedure that extends a given environment with a name/value binding. Finally, `rename` can be implemented as a two-input mapper that discriminates on node type:

```
(define (rename nodes envs)
  (map2D (lambda (node env)
           (cond
            ((var-node? node)
             (var-node (env-lookup (node-name node) env)))
            ((lambda-node? node)
             (lambda-node (env-lookup (node-formal node) env)))
            (else node)))
         nodes
         envs))
```

Behaviorally, the important feature of `alpha-rename` is its space consumption. Define the *working space* of an alpha renaming process to be the maximal space required by the process *excluding* the space required by the input and output terms. Let n be the depth of abstract syntax tree associated with the input term. Then `alpha-rename` has working space *linear* in n. This is the same order of growth as the working space required by the monolithic versions presented in Section 2.2. In contrast, traditional aggregate data implementations of `alpha-rename` would exhibit space *exponential* in n. (See Section 9.2 for experimental results confirming this claim.)

Moreover, disregarding the slag management operations, the fine-grained operational behavior of `alpha-rename` closely resembles that of the functional monolithic version. The lock step nature of SYNAPSE interweaves the processing of the individual slivers into a single left-to-right traversal of the input term. A minor difference between the modular and monolithic processes is that the modular version does an environment lookup for formal parameters as well as variable references; the modular version only performs the lookup for variable references. This difference could be removed by directly wiring a slag from `unique-names` to `rename` and modifying `rename` to use the additional input in the case of lambda nodes.

The Benefits of Modularity

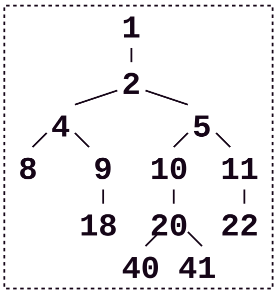
While the sliver-based alpha renamer exhibits important operational characteristics of the monolithic version, it also supports the advantages of modularity. Without these advantages, the work needed to construct all the little pieces that comprise the alpha renamer would be wasted. The real benefit of modularity is that it encourages us to explore alternatives and extensions in a way that the monolithic approach never would. We will now explore a few examples of how modularity and the notion of shape free us to think about the alpha renaming problem from fresh perspectives.

What changes would be required to `alpha-rename` in order to handle lambda terms with extended syntax? The case of handling an `if` construct (see Section 2.2) would be particularly easy: only `term->syndrite` and `syndrite->term` would have to extended with clauses for the new term. `Rename` would not have to be changed since its `else` clause is the appropriate default for simple expressions like `if`. Adding a naming construct like `let` or `letrec` is more challenging. In these cases, `rename` would have to be modified. In addition, the nodes would have to provide `bind` with information indicating in which subtrees the declared name(s) are bound.

The existence of the `unique-names` sliver in the sliver diagrams suggest that there may be many ways to compute unique names. And indeed there are. We can trivially modify the above definition of `unique-names` to visit the formals in a right-to-left direction rather than a left-to-right one. As noted in Section 2.2, such a change can be far from trivial in a monolithic organization.

As a more interesting change, the fact that the name generator is the only sliver in the diagram without a parallel shape suggests that we should consider parallelizing it.[14] One approach to a parallel name generator is to employ the binary tree numbering technique of doubling a number down one branch and doubling and incrementing down the other. This strategy would lead to the following tree for the particular example considered above:

[14]Although we have argued before that networks of purely parallel slivers are probably not a good idea in SYNAPSE, they may be very good idea for other language models. The power of thinking in terms of shapes extends beyond their particular realization in SYNAPSE!



If the branching factor is at most two (as it is in the case of lambda terms), the numbers generated down different branches are guaranteed to be unique, and so they can be calculated in a parallel down fashion. While the `down-scan` sliver is not appropriate for this purpose, it would not be hard to construct a parallel down primitive that is.

Another approach to parallel name generation would be to use `down-scan` with an accumulator that employed a mutable name generator. Of course, the name generator would have to be carefully constructed in order to guarantee atomicity. This is the sort of strategy that is encouraged by Id's M-structures [Bar92].

The alpha renaming network is useful for program manipulations other than alpha renaming. A deBruijn numbering program labels variables in a lambda term with a number indicating the level of the lambda that introduces them [Pey87]. Such a program can be trivially obtained from the alpha renamer by changing the naming strategy from a pre-order scan to a down scan:

```
(define (debruijn names)
  (down-scanD 0
              (lambda (x n) (1+ n))
              names))
```

Furthermore, starting with the deBruijn numbering program, it is easy to obtain a lexical depth compiler. The *lexical depth* of a variable name is a number indicating how many `lambda`s to go up the abstract syntax tree in order to find the one that introduces the name. This number can be computed for each variable reference by subtracting the number found by looking up its name in the environment from the current value of the `debruijn` scanner at variable reference position in the tree. This change requires only (1) an extra slag path between the name generator and `rename` and (2) a modification of the variable reference clause in `rename`.

The fact that the structure of the alpha renamer makes it easy to envision such a wide variety of alternatives, extensions, and related programs is a testament to benefits of modularity. The hope is that the operational control provided by slivers will make such modular organizations even more alluring.

Chapter 7

OPERA: Controlling Operational Behavior

As noted in the exposition of Chapter 5, the sliver technique depends crucially on three language features: concurrency, synchronization, and non-strictness. Concurrency permits the processing of the slivers to be interleaved, synchronization guarantees lock step processing, and non-strictness mechanisms allow fine-grained control over evaluation order in an otherwise strict language. In this chapter, we present OPERA, a particular language that supports these features, and show how they can be used to implement slivers and synchronized lazy aggregates. OPERA is a dialect of Scheme that combines classical mechanisms for concurrency, mutual exclusion, laziness, and eagerness with a novel form of barrier synchronization based on the *synchron*, a new first-class synchronization object.

We begin with an informal introduction to the OPERA language (a formal semantics is deferred until Chapter 8). Then we explore how the features of OPERA can be used to implement the SYNAPSE slivers and slags illustrated in Chapter 6.

7.1 An Introduction to OPERA

OPERA is a dialect of Scheme [ASS85, CR+91] that supports various mechanisms for controlling the fine-grained operational details of programs. Figure 7.1 summarizes the syntax of the language and the primitive operations it supports. Both syntactically and semantically,

Kernel Grammar:

$P \in$ Program
$E \in$ Expression
$I \in$ Identifier
$L \in$ Literal
$S \in$ Symbolic Expression

P	::=	`(program` E_{body} `(define` I_{name} E_{def}`)*)`	[program]
E	::=	L	[literal expression]
	|	I	[variable reference]
	|	`(lambda (`I_{formal}`*)` E_{body}`)`	[abstraction]
	|	`(pcall` E_{proc} E_{arg}`*)`	[parallel application]
	|	`(pletrec ((`I_{name} E_{def}`)*)` E_{body}`)`	[parallel recursive bindings]
	|	`(if` E_{test} E_{then} E_{else}`)`	[conditional]
	|	`(set!` I_{name} E_{val}`)`	[assignment]
	|	`(quote` S_{text}`)`	[quoted expressions]
	|	`(lazon` E_{body}`)`	[suspension]
	|	`(eagon` E_{body}`)`	[future]
	|	`(exclusive` E_{excl} E_{body}`)`	[mutual exclusion]
	|	`(nex ((`I_{name} E_{def}`)*)` E_{body}`)`	[graphical bindings]
	|	`(nexrec ((`I_{name} E_{def}`)*)` E_{body}`)`	[graphical recursive bindings]
I	::=	*usual Scheme identifiers*	
L	::=	*usual Scheme literals*	
S	::=	*usual Scheme quotable expressions*	

Syntactic Sugar Grammar:

E_{sugar} ::= `(cond (`E_{test} E_{action}`*)*)` | `(begin` E_{sequent}`*)` | ... *usual Scheme sugar*
| `(`E_{proc} E_{args}`*)` | `(let ((`I_{name} E_{def}`)*)` E_{body}`)` | `(letrec ((`I_{name} E_{def}`)*)` E_{body}`)`
| `(plet ((`I_{name} E_{def}`)*)` E_{body}`)` | `(seq` E`*)` | `(seq1` E`*)` | `(seqn` E`*)`

Primitives:

OPERA supports the usual Scheme primitive operations on numbers, booleans, procedures, symbols, characters, strings, pairs, vectors, and ports, but it does not support continuations. In addition, it also supports the following objects and operations:

Synchrons: `(synchron)`, `(wait` E_{sync}`)`, `(simul!` E_{sync1} E_{sync2}`)`,
`(precede!` E_{before} E_{after}`)`, `(synchron?` E_{obj}`)`

Excludons: `(excludon)`, `(excludon?` E_{obj}`)`

Cells: `(cell` E_{val}`)`, `(cell-ref` E_{cell}`)`, `(cell-set!` E_{cell} E_{new}`)`, `(cell?` E_{obj}`)`

Figure 7.1: OPERA summary

OPERA is a close cousin of Scheme. The key differences are as follows:

- *Concurrency*: OPERA supports two forms of concurrency:

  1. concurrent evaluation of the subexpressions of a procedure call expression.
  2. eager evaluation via the `eagon` special form.

  Here we briefly describe the concurrent evaluation strategy for procedure calls; eagons are described in the bulleted item on non-strictness.

  Both OPERA and Scheme have strict procedure calls, but they use different strategies for evaluating the subexpressions of a procedure application expression: OPERA evaluates the subexpressions in parallel, while Scheme requires that the subexpressions be evaluated in a way that is consistent with some sequential order. The explicit `pcall` (*p*arallel *call*) keyword in OPERA's kernel application form is intended to emphasize this difference. However, since the explicit keyword is cumbersome, we will abbreviate `(pcall` E_{proc} E_{arg}`*)` as the more familiar `(`E_{proc} E_{arg}`*)`. OPERA's concurrent evaluation strategy extends to the bindings of `plet` and `pletrec`, parallel versions of Scheme's `let` and `letrec`. To simplify comparisons between the languages, OPERA treats `let` and `letrec` as synonyms for `plet` and `pletrec`.

  While OPERA programs are designed to look like Scheme programs, OPERA's concurrent evaluation strategy can lead to behaviors that would not be exhibited by a Scheme interpreter. In particular, the concurrency can give rise to new kinds of nondeterminism. For this reason, it is important to clearly specify whether a program in Scheme syntax is to be evaluated with respect to OPERA or to Scheme.

  We emphasize that OPERA supports concurrency purely for the purpose of modularity, not for efficiency. Multi-tasking on a single physical processor is a sufficient form of concurrency for our purposes.

- *Synchronization*: OPERA supports two first-class synchronization objects: synchrons and excludons. Synchrons are the key to the lock step processing model on which slivers and synchronized lazy aggregates are based. They provide barrier synchronization in a system where the number of processes participating in a rendezvous is determined dynamically.

Excludons are just standard lock objects [Bir89b] that support mutual exclusion via the `exclusive` special form. Excludons are used to enforce atomic actions by eliminating the undesirable nondeterminism introduced by concurrency.

- *Non-strictness*: OPERA supports lazy and eager evaluation via the `(lazon` E_{body}`)` and `(eagon` E_{body}`)` special forms. Both forms create a placeholder object that stands for the result of evaluating E_{body}. In the case of lazons, evaluation of E_{body} is delayed until the lazon appears in a context that requires its value. Lazons are like the *delayed objects* described in [Hen80] and [ASS85] except that they are implicity forced rather than explicitly forced; we shall see how this difference enhances modularity. In the case of eagons, evaluation of E_{body} proceeds in parallel with the rest of the computation; any context requiring the value of E_{body} must wait until it is available. Eagons are the same as the *futures* used in various parallel Lisp implementations ([Hal85], [Mil87]), except that in OPERA we do not assume the existence of physical parallelism. We use the name "eagon" only for symmetry with "lazon".

- *Other differences*: OPERA supplies a few additional features that add no new power to the language but simplify the expression of some programs. OPERA introduces two new binding forms: `nex` and `nexrec`. These are versions of `let` and `letrec` that do not evaluate the binding expressions before evaluating the body. Instead, they resemble the "graphical" `let`s and `letrec`s of a lazy functional language (e.g., see [Pey87, pages 233-234]). In addition to the usual Scheme syntactic sugar, OPERA provides a few new derived special forms. There are three sequencing forms: `(seq` $E*$`)`, `(seq1` $E*$`)`, and `(seqn` $E*$`)`. All evaluate their subexpressions in left-to-right order, but they return different results: `seq` returns the boolean true value, `seq1` returns the value of the first subexpression, and `seqn` returns the value of the last subexpression (i.e., it is a synonym for `begin`). In addition to the usual Scheme datatypes and operations, OPERA supports mutable one-slot data structures called *cells*. These correspond to the reference structures of ML [MTH90] and FX [GJSO92].

> OPERA does not support Scheme's continuations. In principle, OPERA should ultimately be able to support continuations, but the details of how continuations interact with OPERA's fine-grained concurrency have not yet been worked out.

The summary in Figure 7.1 gives the form of an OPERA program as:

$$\texttt{(program } E_{\text{body}} \texttt{ (define } I_{\text{name}}\ E_{\text{def}}\texttt{)*)}$$

In our OPERA examples below, we will not use the `program` form directly. Instead, in the usual Scheme style, we will treat every expression as a program by appropriately packaging it with all the definitions encountered so far.

The following sections explore the features of OPERA in more depth.

7.1.1 Strict Procedure Calls

As in Scheme, the procedure call of OPERA is strict. This means that the evaluation of the procedure body does not begin until the evaluation of the arguments has completed. Because it provides natural barrier between the computation of the arguments and the computation of the body, OPERA's strict procedure call serves as a basic mechanism for controlling operational behavior. Lazons and eagons are mechanisms for overriding the strictness of a procedure call.

7.1.2 Concurrent Evaluation

To permit the concurrent evaluation of slivers, OPERA embraces a concurrent evaluation strategy for the subexpressions of a procedure application. Synchrons and excludons are mechanisms for controlling the concurrency introduced by this strategy.

In this context, "concurrent" does not imply parallel execution of a program on multiple physical processors. It merely indicates a partial ordering on the operations performed in the evaluation of the subexpressions of a procedure call. That is, the evaluation steps of a procedure call's subexpressions may be arbitrarily interleaved. (A precise model of the allowed interleaving is provided in Chapter 8.) Whether the interleaving results from multitasking on a single processor or from evaluating the subexpressions on different processors is irrelevant for our purposes. We use concurrency not to make programs run faster, but to make them more modular. In fact, this work clearly supports the notion that concurrency is an important modularity mechanism. Although orchestrating a program on a parallel machine fits into the general theme of controlling the operational behavior of modular programs, it is beyond the scope of this work.[1]

As indicated in the overview, the concurrent evaluation strategy is a fundamental feature distinguishing OPERA from Scheme. To compare the languages, consider the following definitions:

```
(define (par a b) #t)

(define (test-order)
  (par (par (display 1)
            (display 2))
       (par (display 3)
            (display 4))))
```

The `par` procedure ignores the values of its two arguments and returns the boolean *true* value (`#t`). In both Scheme and OPERA, argument expressions are always evaluated regardless of whether their values are actually used by the procedure. So (`par` E_1 E_2) has the effect of evaluating both E_1 and E_2. `Test-order` is a procedure that tests argument evaluation order by using `par` in conjunction with `display`, a printing procedure.[2]

Because OPERA evaluates all application subexpressions in parallel, there are no constraints on the execution order of the `display`s within `test-order`. Evaluating `(test-order)` in OPERA may print the numerals `1` through `4` in any of the 24 possible orderings. In Scheme, on the other hand, the possible behaviors of this expression are more limited. Scheme evaluates all subexpressions of an application in some (unspecified) sequential order. This means that one of the arguments of `test-order`'s outermost `par` must be completely evaluated before evaluation of the other is begun. For example, `1` may be printed before both the `3` and `4` or after both the `3` and `4` but never between the `3` and `4`. Scheme allows only eight possible orderings for the numbers displayed by `(test-order)`:

[1] See [HS86b] for work along these lines.

[2] For simplicity, we will often illustrate issues of evaluation order with contrived examples that involve I/O or assignment. However, in practice, we are more interested in the relative order of purely functional operations. This order provides a window on the space consumed by pending operations.

`1234, 1243, 2134, 2143, 3412, 3421, 4312, 4321`

It is worth noting that OPERA is not as concurrent as it could be. Although it exhibits concurrency in argument evaluation, there is no concurrency between the evaluation of a procedure's arguments and the evaluation of the procedure's body. So OPERA, like Scheme, is a *strict* language — i.e., it requires all arguments to be evaluated before the body is evaluated. For example, consider the following:

```
(define (ignore a) (display 2))

(define (test-strict) (ignore (display 1)))
```

The call `(test-strict)` is guaranteed to print `1` before `2` in both OPERA and Scheme. Strictness effectively defines a barrier between the evaluation of the arguments and the evaluation of the body. This barrier is crucial for controlling the behavior of OPERA programs. We'll see shortly how non-strict behavior can be obtained in OPERA with lazons, and eagons.

Concurrent evaluation and strictness characterize OPERA's `let` (= `plet`) and `letrec` (= `pletrec`) constructs as well. That is, in both cases the binding expressions are evaluated in parallel, but all of these must be completely evaluated before the body is evaluated.

A final subtlety is the extent to which a language implementation must exhibit permitted evaluation orderings in practice. For example, a particular Scheme implementation is *not* required to exhibit all eight of the possible orderings for the `(test-order)` example. In fact, a typical Scheme interpreter follows a fixed ordering that is consistent with the "some sequential order" rule. The rule is more often exploited by Scheme compilers, which can permute the argument evaluation order as convenient. In fact, a Scheme implementation may actually interleave evaluation of arguments as long as it can prove that the resulting behavior is consistent with the "some sequential order" rule (e.g., when the argument expressions have no side effects).

What about OPERA? In the presence of synchronization mechanisms like synchrons and excludons, artificially limiting the range of evaluation orders can prevent a program from terminating with an expected result via one of the permitted orderings. So any restricted set of evaluation orders is generally unacceptable. Nevertheless, an implementation may restrict the evaluation order in contexts where it can prove that the restriction will not affect normal termination of a program.

7.1.3 Synchrons

Concurrency alone does not guarantee that slivers networks will exhibit desirable space profiles and operation orderings. After all, a producing sliver might race ahead of one of its consumers, requiring the generated elements to be buffered. To avoid this situation, the sliver technique is based on a lock step processing model. Lock step processing is achieved via synchrons.

Synchronization Model

A *synchron* is a first-class object that represents a point in time. Computational events can be constrained to happen before or after this point in time, or left unordered with respect to it. Synchrons are used to express control in a concurrent system by limiting the allowable orderings between computational events. Intuitively, synchrons express a set of time constraints that are solved by OPERA.

The lock step model of the sliver technique implies that slivers are concurrently executing recursive procedures that participate in a *barrier synchronization* at every corresponding call and return. 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 configurability of sliver networks, the number of slivers participating in a barrier synchronization cannot generally be predicted from the text of the program. For example, the number of slivers created by the `pascal` procedure on page 243 depends on the values of its `column` argument.

Synchrons solve this problem by tying 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 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. *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. The automatic storage manager is responsible for determining when a rendezvous has been achieved at a synchron and resuming the waiting processes once the rendezvous has occurred. Because all processes lose access to the synchron upon resumption, there can only be one rendezvous per synchron.

The rendezvous protocol of synchrons sets it apart from other synchronization structures (e.g., semaphores [Dij68], locks [Bir89b], 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 when a global rendezvous state is achieved. In this respect, synchrons resemble *weak pairs* [Mil87] and *populations* [RAM83], data structures whose behavior is tied to storage management.

Although developed independently, synchrons are closely related to Hughes's `synch` construct [Hug83, Hug84]. (See Section 3.1.5 for a discussion of `synch`.) Both `synch` and synchrons involve objects that define a rendezvous point. Hughes's objects serve as a rendezvous for exactly two requests for a value. It is possible to build a tree of `synch`s that act as a rendezvous point for a given number of requests, but these trees must be explicitly managed by the programmer. It is unclear how `synch`s could be used to handle a dynamically determined number of requests; I doubt that they can be used for this purpose. In contrast, synchrons serve as a rendezvous point for a dynamically determined number of processes. Because they automatically manage the barrier synchronization of an arbitrary number of processes, synchrons are superior to `synch` for building synchronization abstractions. Furthermore, the additional unification and precedence operations supported by synchrons (see below) are crucial for implementing important aspects of the lock step processing model.

Synchron Operations

Figure 7.2 summarizes the procedural interface to synchrons. The nullary `synchron` procedure creates a new synchron. Since synchrons are first-class objects, they can be named, passed as arguments to procedures, returned as results from procedures, and stored in data strucutures. The `synchron?` predicate tests whether an object is a synchron.

`(synchron)`
: Creates a new synchron.

`(synchron?` *obj*`)`
: Returns a boolean value indicating whether *obj* is a synchron.

`(wait` *sync*`)`
: Waits for a rendezvous at the synchron *sync*. Returns `#t` after the rendezvous has occurred.

`(simul!` *sync1 sync2*`)`
: Unifies synchrons *sync1* and *sync2* to be the same synchron. Returns the unified synchron.

`(precede!` *sync1 sync2*`)`
: Dictates that a rendezvous at the synchron *sync1* must occur before a rendezvous at the synchron *sync2*. Returns `#t`.

Figure 7.2: The procedural interface to synchrons.

The key operation on synchrons is the `wait` procedure. Calling `wait` on a synchron *sync* suspends the current computation until a rendezvous occurs at *sync*. After the rendezvous, all computations waiting on the synchron are resumed with the return of `#t` to each of the `wait` calls.

A `wait` call holds the argument synchron with a distinguished waiting pointer; all other references to the synchron are non-waiting pointers. The rendezvous occurs when the synchron is held only by waiting pointers. Since the return from `wait` drops all of the waiting pointers, a synchron is inaccessible after a rendezvous and its storage may be reclaimed.

The `simul!` procedure declares that the rendezvous of one synchron must occur simultaneously with the rendezvous of another synchron. Conceptually, `simul!` extends the notion of unifying logic variables [Rob65] to unifying the points in time represented by two synchrons. The `simul!` procedure forces its two argument synchrons to be equivalent (as determined by OPERA's `eq?` predicate); it returns the unified synchron as its result.

The `precede!` procedure declares that a rendezvous on one synchron must precede a rendezvous on another synchron. `Precede!` is an explicit method of imposing an ordering on the points in time represented by synchrons. (Synchrons can also be ordered implicitly by the context in which they are used.)

`Wait`, `simul!`, and `precede!` can be viewed as specifying a set of ordering constraints on the events of a computation. OPERA "solves" the set of constraints by dynamically finding an event ordering that satisfies the constraints. Sometimes the constraints are incompatible and no solution exists. For example, no ordering of events can satisfy the declaration that a synchron a precedes itself, or that a precedes b and b precedes a. If OPERA is forced to resolve an incompatible set of constraints, it halts in a *deadlock* state.

Synchron Examples

We illustrate the basics of synchrons through a series of simple examples. As in our discussion of the concurrent evaluation strategy, we will consider only examples involving the ordering of simple I/O operations. The discussion of the SYNAPSE implementation in Section 7.2 provides more realistic examples of synchron use.

We begin with the `test-order` procedure from Section 7.1.2:

```
(define (test-order_initial)
  (par (par (display 1)
            (display 2))
       (par (display 3)
            (display 4))))
```

In OPERA, (`test-order`$_{initial}$) prints out the numerals `1` through `4` in arbitrary order; there are 24 possibile orderings.

Suppose we want to specify the constraint that `3` must be printed before `2`. We can achieve this effect by rewriting the procedure as:

```
(define (test-order_reorganize)
  (par (display 1)
       (par (seq (display 3) (display 2))
            (display 4))))
```

The sequencing of seq implements the desired constraint.[3] However, this approach forces us to significantly reorganize the structure of the original procedure.

An alternate approach for expressing the constraint is to use a synchron:

```
(define (test-order_synchron:3->2)
  (let ((a (synchron)))
    (par (par (display 1)
              (seq (wait a) (display 2)))
         (par (seq (display 3) (wait a))
              (display 4)))))
```

The (synchron) call returns a new synchron object, which is named a in this example. The two (wait a) expressions inserted into the program implement the desired constraint by requiring 3 to be printed before the rendezvous on a and 2 to be printed after the rendezvous.

The advantage of the synchron approach over the approach taken in test-order$_{reorganize}$ is that the basic structure of the procedure (a par of two pars) is unchanged. The creation of the synchron and the insertions of the seqs and waits are essentially annotations that have been added to the original structure. This annotation-like character of synchrons is used to preserve modularity while expressing control.

As another example, imagine constraining test-order$_{initial}$ to print 4 after both 1 and 3. This is also straightforward to do with a synchron:

```
(define (test-order_synchron:1+3->4)
  (let ((b (synchron)))
    (par (par (seq (display 1) (wait b))
              (display 2))
         (par (seq (display 3) (wait b))
              (seq (wait b) (display 4))))))
```

Here, two events occur before the rendezvous at the synchron, and one occurs after. In general, an arbitrary number of waits can be used to specify constraints between events. While the given constraints can also be expressed by using the right permutation of par and seq, the reorganized program would not carry with it any artifact of how it was derived by constraining a simpler program. In contrast, synchrons can be added and removed without altering the basic structure of the program.

[3] Recall that seq is an OPERA special form whose subexpressions are evaluated sequentially from left to right. Together, par and seq constitute a language for the series/parallel combination of expressions.

Another advantage of synchrons is that they facilitate the combination of timing constraints. Here's a single program that embodies the ordering constraints of the previous two examples:

```
(define (test-order_synchron:3->2&1+3->4)
  (let ((a (synchron))
        (b (synchron)))
    (par (par (seq (display 1) (wait b))
              (seq (wait a) (display 2)))
         (par (seq (display 3) (par (wait a) (wait b)))
              (seq (wait b) (display 4))))))
```

Note how the synchron annotations of the previous programs have essentially been superposed in this one. The program permits only five orderings of displayed numerals: `1324`, `1342`, `3214`, `3124`, and `3142`. It is impossible to express this set of orderings with any combination of `par`s and `seq`s. [4]

We will use the following `test-combiner` procedure to illustrate the different means of combining synchrons:

```
(define (test-combiner combiner)
  (let ((c (synchron))
        (d (synchron)))
    (seq (combiner c d)
         (par (seq (display 1) (seq (wait c) (display 2)))
              (seq (display 3) (seq (wait d) (display 4)))))))
```

`Test-combiner` first combines the synchrons `c` and `d` (by side effect), and then executes two processes: one of which prints `1` before `2` and the other of which prints `3` before `4`. Synchron `c` represents the point between the printing of `1` and `2`, while `d` represents the point between the printing of `3` and `4`.

We consider three combiners:

1. `(test-combiner (lambda (x y) 'ignore))` : Here the combiner is a trivial one that leaves `c` and `d` unconstrained. In this case there are six possible outputs:

   `1234, 1324, 1342, 3124, 3142, 3412`

[4] To be fair, neither `par`/`seq` nor synchrons are sufficient to express many orderings. For example, neither is sufficient to express the set $\{1234, 4321\}$, which requires some additional form of nondeterminism. But OPERA with synchrons can express more orderings than OPERA without synchrons.

2. `(test-combiner precede!)` : The `precede!` combiner forces the `c` rendezvous to occur before the `d` rendezvous. This has the effect of adding the constraint that the `1` be printed before the `4`. This filters out the `3412` option, leaving only five possible behaviors:

   `1234`, `1324`, `1342`, `3124`, `3142`

3. `(test-combiner simul!)` : The `simul!` combiner forces the `c` rendezvous and `d` rendezvous to occur simultaneously. This implies two additional constraints over the unconstrained version: `1` must be printed before `4`, and `3` must be printed before `2`. This leaves four possible behaviors:

   `1324`, `1342`, `3124`, `3142`

All of the above examples illustrate how synchrons can be used to constrain the order of operations within a single procedure body. But the real power of synchrons is that they can be used to constrain the order of operations across procedure boundaries. For example, consider the following procedure, which abstracts over the notion of inserting a synchronization point between two `display`s:

```
(define (make-displayer a b)
  (lambda (sync)
    (seq (display a) (seq (wait sync) (display b)))))
```

Using `make-displayer`, the three combination examples from above can be expressed in a modular fashion as follows:

1. No constraints – pass independent synchrons to two displayers:

   ```
   (par ((make-displayer 1 2) (synchron))
        ((make-displayer 3 4) (synchron))
   ```

2. `Precede!` constraint – pass two constrained synchrons to the two displayers:

   ```
   (let ((c (synchron))
         (d (synchron)))
     (seq (precede! c d)
          (par ((make-displayer 1 2) c)
               ((make-displayer 3 4) d))))
   ```

3. `Simul!` constraint – pass the same synchron to both displayers (no need to actually unify two separate synchrons via `simul!`):

```
(let ((c (synchron)))
  (par ((make-displayer 1 2) c)
       ((make-displayer 3 4) c)))
```

As a final example, we will illustrate how synchrons can be stored in data structures. This fact is especially important for implementing synchronized lazy aggregates. We will use the following procedure to test this feature:

```
(define (make-synchronized-displayer a b)
  (let ((sync (synchron)))
    (cons sync
          (lambda ()
            (seq (display a) (seq (wait sync) (display b)))))))
```

`Make-synchronized-displayer` is similar to the `make-displayer` procedure from above. However, rather than returning a synchron-accepting procedure, it returns a pair of a synchron and a thunk that refers to that synchron. The synchron is a hook for controlling some behavioral aspects of the associated thunk. Here is an expression that yields the behavior of the `simul!`-constrained versions of the previous examples (the other two constraints can be expressed similarly):

```
(let ((sync-disp1 (make-synchronized-displayer 1 2))
      (sync-disp2 (make-synchronized-displayer 3 4)))
  (seq (simul! (car sync-disp1) (car sync-disp2))
       (par ((cdr sync-disp1)) ((cdr sync-disp2)))))
```

This example underscores how independently generated synchronization points can be unified together to constrain the behavior of concurrently executing procedures. This is exactly the mechanism used to guarantee the lock step processing of slivers networks exhibiting fan-in.

Some Details

We conclude the overview of synchrons by mopping up a few details:

- *Deadlock*: None of the above examples illustrate deadlock, but it's easy to exhibit examples that do. Here are a number of expressions that give rise to deadlock:

```
;; Example 1
(let ((a (synchron)))
  (seq (wait a) (wait a)))


;; Example 2
(let ((a (synchron))
      (b (synchron)))
  (seq (unify! a b)
       (seq (wait a) (wait b))))


;; Example 3
(let ((a (synchron))
      (b (synchron)))
  (par (seq (wait a) (wait b))
       (seq (wait b) (wait a))))


;; Example 4
(let ((a (synchron))
      (b (synchron)))
  (seq (precede! b a)
       (seq (wait a) (wait b))))


;; Example 5
(let ((a (synchron)))
  (seq (wait a) a))


;; Example 6
(let ((a (sychron)))
  (if (wait a)
      17
      a))
```

Examples 1 and 2 deadlock because they require a rendezvous to happen before itself. Similarly, the deadlocks in examples 3 and 4 are due to cycles in the time ordering constraints between two synchrons. The expression in example 5 deadlocks because the second `a` in `(seq (wait a) a)` is a non-waiting reference to the synchron that prevents the rendezvous from proceeding; since the reference to the second `a` is not dropped until after `wait` successfully returns, there is a constraint cycle here as well. Example 6 is an extension to example 5. It is somewhat disconcerting that this expression deadlocks; because `wait` always returns `#t`, the `a` in the else branch of the `if` can never be reached anyway. A "sufficiently smart" implementation of OPERA could determine this fact through static analysis. However, we will rely only on the semantics provided in Chapter 8, which dictate that example 6 must also deadlock.

A careful reader familiar with the Scheme environment model [ASS85] may wonder why a lot *more* examples don't deadlock. Consider the following (non-deadlocking) OPERA expression:

```
(let ((a (synchron))
      (b 17))
  (seq (wait a) (+ b 2)))
```

Under the traditional environment model, the expression `(seq (wait a) (+ b 2))` is evaluated in an environment in which `a` is bound (via a non-waiting pointer) to a synchron and `b` is bound to 17. Since the environment must be present for the evaluation of `(+ b 2)`, and it contains a non-waiting pointer to a synchron, this expression *would* deadlock under the traditional environment model. In the environment model, any expression that names a synchron could deadlock in this way; synchrons are rather useless in such a model.

The reason that such expressions *don't* deadlock is that OPERA is defined in terms of a model of evaluation that aggressively drops spurious references. Since the binding of `a` is not required to evaluate `(+ b 2)`, the non-waiting pointer to the synchron is effectively removed from the environment after the `a` in `(wait a)` is evaluated. The semantics in Chapter 8 spells out how this is accomplished.

- *Blocking behavior of non-waiting synchron references*: The above examples used the idiom

  `(seq` E `(wait` *sync*`))`

  to evaluate E before the rendezvous of *sync*. A similar effect can be achieved by the simpler idiom:

  `(seq` E *sync*`)`

The reason the latter version works is that *sync* is a non-waiting synchron reference that is not dropped until after E is evaluated ; such a reference serves to block a rendezvous on the synchron until it is dropped.

In fact, the `wait`-less version is often preferable to the `wait` version because it implies fewer constraints. Consider the following related test procedures:

```
(define (test-waitful)
  (let ((a (synchron)))
    (par (par (seq (seq (display 1) (wait a)) (display 2))
              (seq (seq (display 3) (wait a)) (display 4)))
         (seq (wait a) (display 5)))))

(define (test-waitless)
  (let ((a (synchron)))
    (par (par (seq (seq (display 1) a) (display 2))
              (seq (seq (display 3) a) (display 4)))
         (seq (wait a) (display 5)))))
```

`Test-waitful` uses the `wait` idiom for expressing a before constraint, while `test-waitless` uses the `wait`-less idiom. Both procedures specify the constraint that both `1` and `3` are printed before `5`. However, `test-waitful` also constrains `1` to be printed before `4` and `3` to be printed before `2`. In contrast, `test-waitless` does not express these extra constraints.

The blocking behavior non-waiting references has other consequences as well. Reconsider the last version of the `simul!`-constrained examples:

```
(let ((sync-disp1 (make-synchronized-displayer 1 2))
      (sync-disp2 (make-synchronized-displayer 3 4)))
  (seq (simul! (car sync-disp1) (car sync-disp2))
       (par ((cdr sync-disp1)) ((cdr sync-disp2)))))
```

Question: does replacing the `seq` in this example by a `par` change the behavior of the code? It tempting to answer "yes", with the rationale that the unification could possibly happen after all the `display`s if it were allowed to proceed in parallel with them. In fact, the answer is "no". The reason is that the `simul!` expression holds non-waiting pointers to both synchrons in question. The `wait`s peformed by the thunks can't possibly return until the non-waiting pointers are dropped, which is only after the unification has succeeded. There is no danger of a race condition here.

That's the good news. The bad news is that the blocking behavior of non-waiting pointers has an nasty side as well. In particular, it can easily lead to deadlock. Consider the following innocuous-looking expression:

```
;; Deadlocking version
(let ((pair (cons (synchron) 17)))
  (seq (wait (car pair))
       (+ 2 (cdr pair))))
```

This expression is guaranteed to deadlock! The reason is that the pair value of `pair` holds a non-waiting pointer to the synchron and that pointer is accessible until `(cdr pair)` is evaluated. But `(cdr pair)` cannot be evaluated until the `wait` on the synchron returns. This subtle dependency cycle leads to deadlock.

In this case the deadlock can be avoided by forcing the `cdr` to be performed before the `wait`:

```
;; Non-deadlocking version that returns 19
(let ((pair (cons (synchron) 17)))
  (let ((num (cdr pair)))
    (seq (wait (car pair))
         (+ 2 num))))
```

In fact, when dealing with a synchron-bearing data structure, it's always a good idea to unbundle the structure into its components before `wait`ing on any of the synchrons. This *aggressive unbundling* strategy is required of every sliver (see Section 5.4.2) precisely to avoid spurious deadlocks.

- *Is* `precede!` *necessary?*: With the introduction of eagons (see below), `precede!` seems superfluous because it could be defined as:

```
(define (precede! sync1 sync2)
  (eagon (seq (wait sync1) (wait sync2))))
```

This definition uses `eagon` to fork off a process that requires the rendezvous on the first argument to occur before the the rendezvous on the second argument.

Nevertheless, it turns out that a primitive `precede!` form allows certain garbage collection subtleties to be resolved in a pleasing way that is not readily available with the user-defined version (see Section 8.2.2 for details). For this reason, we stick with `precede!` as a primitive.

7.1.4 Excludons

The rendezvous style of synchronization supported by synchrons is helpful for getting parts of a computation to work in lock step. However, synchrons don't help to express *mutual exclusion* between parts of a computation. Constraining OPERA's concurrent argument evaluation strategy to be like Scheme's some-sequential-order strategy is an example of mutual exclusion: the order in which the arguments are evaluated does not matter, but each argument evaluation occupies a single interval of time that cannot intersect with the others.

For expressing mutual exclusion, OPERA supports *excludon* objects. An excludon serves as a key that is needed to gain access to various regions of a computation; because its use is restricted to one region at a time, it guarantees that the regions execute in mutually exclusive time intervals. Excludons are similar to the locking mechanisms supplied in many concurrent systems (e.g. semaphores [Dij68], locks [Bir89b], monitors [Hoa74]).

Each call to the nullary `excludon` constructor creates a unique, first-class excludon object. The form `(exclusive` E_{excl} E_{body}`)` first evaluates E_{excl}, which should be an excludon x, and then evaluates E_{body} while having exclusive hold on x. If x is already held by some other `exclusive`, then evaluation of E_{body} blocks until x is released.

Excludons are used to guarantee that sequences of operations behave as atomic actions. This is particularly important when the operations involve side effects. For example, suppose $\{a, b\}$ represents a time interval in which both a and b are printed (in some order). Then the following procedure guarantees that $\{\texttt{1}, \texttt{2}\}$ must precede $\{\texttt{3}, \texttt{4}\}$ or follow it:

```
(define (test-order_exclusive)
  (let ((e (excludon)))
    (par (exclusive e
           (par (display 1)
                (display 2)))
         (exclusive e
           (par (display 3)
                (display 4))))))
```

Excludons can be used to simulate Scheme's some-sequential-order argument evaluation within OPERA. Consider the `scall` special form, defined by the following desugaring:

```
(scall E_proc E_arg1 ... E_argn)
  desugars to
(let ((I_excl (excludon)))                  ; I_excl is a fresh variable
  (pcall (exclusive I_excl E_proc)
         (exclusive I_excl E_arg1)
                          ⋮
         (exclusive I_excl E_argn)
         ))
```

The excludon named I_{excl} prohibits interleaved evaluation among the subexpressions of the `pcall`. Sequential versions of `let` and `letrec` (call them `slet` and `sletrec`) can also be defined by restricting the corresponding parallel form with excludons.

7.1.5 Non-strictnesss

In OPERA, evaluation of `pcall` arguments is strict: all arguments must evaluate to a value before the procedure can be applied. This is not always the desired semantics. Sometimes we would like an argument not to be evaluated until it is actually used in the body. That way, if the argument is never used, it will never be evaluated. This strategy is called *laziness*. On the other hand, we sometimes want the body and the arguments evaluated in parallel, regardless of whether body actually uses the arguments. This strategy is called *eagerness*. Both laziness and eagerness are examples of *non-strict* evaluation strategies.

OPERA supports these non-strict evaluation strategies because they are helpful for controlling how computations unfold. There are two types of objects used for this purpose: *lazons* and *eagons*. Lazons support lazy evaluation, while eagons support eager evaluation.

Lazons

A *lazon* is a first-class object that serves as a placeholder for the value of an expression. Lazons are created by the special form (`lazon` E_{body}). The evaluation of E_{body} is delayed until its value is demanded by the computation. A context that requires the value of a lazon's body is said to *touch* the lazon. Examples of touching contexts are argument positions of an arithmetic operator, the test position of an `if`, and the operator position of a `pcall`. All of these contexts need to know specific details about the value of E_{body}. Non-touching contexts, such as the operand positions of a `pcall` and the argument positions of a `cons`, do not probe any details of the lazon's body, and simply pass around the lazon instead. Lazons are similar to the delayed values described in [Hen80] and [ASS85], except that delayed values are touched by application of an explicit `force` procedure. [5] In OPERA, the forcing is done implicitly by a touching context.

Consider the following definitions:

```
(define (display&return x) (begin (display x) x))

(define (test-strict2 a b) (begin (display 3) (* b b)))
```

Then in OPERA

```
(test-strict2 (display&return 1) (display&return 2))
```

prints a `1` and `2` in some order before printing a `3` and returning `4`. Note that `(display&return 1)` is evaluated even though its value is not used by the procedure. In contrast,

```
(test-strict2 (lazon (display&return 1)) (lazon (display&return 2)))
```

prints a `3` followed by a `2` before returning `4`. Because lazons are objects, they satisfy the strictness requirement of the parallel call without their bodies being evaluated. The expression `(display&return 2)` is not evaluated until the value of `b` is demanded by the `*`, which happens only after the `3` has been printed. Even though the value of `b` is demanded twice, a `2` is printed only once because the value of a lazon is *memoized* – the lazon body is evaluated once, and its value is cached for subsequent touches. The `1` is never printed because the formal parameter `a` of `test-strict2` is never referenced in the body.

If there is ever a need to explicitly force a lazon, this can be done with the `touch` procedure, defined as follows:

```
(define (touch x) (if x x x))
```

[5] The Scheme report [CR$^+$91] permits, but does not require, implicit forcing of the promises created by Scheme's `delay`. In contrast, OPERA requires implicit touching of lazons.

The test position of the `if` is a touching context, so it forces the evaluation of the lazon body; that value is returned regardless of whether it is false or not.[6] Memoization guarantees that the evaluation happens only once. `Touch` acts as the identity procedure on non-lazons.

OPERA does not support any predicate for testing whether an object is a lazon. No program can ever distinguish whether or not a purely functional expression has been wrapped in a lazon. If an expression performs side effects, however, then wrapping the expression in a lazon may prevent those side effects from occurring in some contexts.

Lazons can be used to define a lazy argument evaluation strategy. Consider the `lcall` special form, defined by the following desugaring:

(lcall E_{proc} E_{arg_1} ... E_{arg_n})
  desugars to
(pcall E_{proc}
    (lazon E_{arg_1})
      ⋮
    (lazon E_{arg_n})
)

The `lazon`s annotating the arguments prevent them from being evaluated before the procedure is called. The arguments will only be evaluated if they are needed within the body of the called procedure. `Lcall` corresponds to the *call-by-need* parameter passing mechanism common in functional programming languages (e.g. Haskell [HJW+92] and Miranda[Tur85]).

Like the delayed objects of [Hen80] and [ASS85], lazons can be used to create conceptually infinite data structures and finesse certain kinds of circular dependencies. For example, the stream datatype of [ASS85] can be defined in OPERA as follows:

```
;;; Syntactic sugar:
;;; (cons-stream E_head E_tail)
;;;    desugars to
;;; (cons E_head (lazon E_tail))

;;; Procedures
(define head car)
(define tail cdr)
(define the-empty-stream '())
(define empty-stream? null?)
```

[6] In OPERA, as in Scheme, tests of a conditional need not be boolean objects. Any non-false object is treated as true.

Note that OPERA's `tail` operation, unlike Scheme's, does not force the evaluation of the tail of the stream. OPERA relies on touching contexts to do this forcing.

The implicit forcing of lazons supports modularity better than the explicit forcing of delayed objects. We will illustrate this point using a stream integration problem described in [ASS85]. Integration can be defined by the following elegant procedure:

```
(define (integral_original integrand initial-value dt)
  (letrec ((int (cons-stream
                 initial-value
                 (add-streams (scale-stream dt integrand)
                              int))))
    int))
```

`Integral` takes a stream of values, an initial value, and a time step, and returns a stream of the running sum of scaled values (where the sum is initialized to `initial-value`).

Unfortunately, certain feedback problems cannot use this `integral` procedure. For example, consider the following procedure for solving the equation $dy/dt = f(y)$ by integration:

```
(define (solve f y-init dt)
  (letrec ((y (integral dy y-init dt))
           (dy (map-stream f y)))
    y))
```

The `solve` procedure fails because of a circular dependency between `y` and `dy` that is unsolvable (as expressed) in a strict language. The solution suggested in [ASS85] is to delay the first argument to `integral`:

```
(define (solve f y-init dt)
  (letrec ((y (integral (delay dy) y-init dt))
           (dy (map-stream f y)))
    y))
```

But because delayed objects must be explicitly forced, this necessitates a change to `integral`:

```
(define (integral_delayed delayed-integrand initial-value dt)
  (letrec ((int (cons-stream
                 initial-value
                 (add-streams (scale-stream dt (force delayed-integrand)) ; ***
                              int))))
    int))
```

Delaying an argument to a procedure has the undesirable effect of changing its interface. This requires modifying not only the body of the procedure, but all calls to the procedure as well. This is a very non-modular approach to delayed evaluation.

Lazons are much more suitable for such problems. With lazons, the circularity of `solve` can be circumvented as follows:

```
(define (solve f y-init dt)
  (letrec ((y (integral (lazon dy) y-init dt))
           (dy (map f y)))
    y))
```

Because lazons are implicitly touched, there is no need to modify the definition of $\texttt{integral}_{original}$, nor any of the existing calls to `integral`. This example illustrates why lazons are superior to delayed objects for preserving modularity.

Eagons

Like lazons, eagons are first-class placeholders for the value of an expression. But whereas a lazon delays the evaluation of its body until it is demanded, an eagon may begin the evaluation of its body as soon as it is created. Eagons are implicitly touched by the same contexts as lazons. Touching an eagon blocks the current computation until the eagon body has been fully evaluated, at which point the computation resumes with this value. Those familiar with *futures* ([Hal85], [Mil87],[For91]) will recognize that "eagon" is just another name for "future".

Eagons are created by the special form (`eagon` E_{body}). Here's a sample use of eagons as arguments to the `test-strict2` procedure from page 316:

```
(test-strict2 (eagon (display&return 1)) (eagon (display&return 2)))
```

The argument expressions immediately evaluate to eagons, allowing the body of `test-strict2` to be evaluated in parallel with the two calls to `display&return`. This means that `1`, `2`, and `3` can be printed in any order. Even though the first argument is never touched, eagons always evaluate their bodies, so the `1` is printed anyway. In fact, the lack of dependences involving `(display&return 1)` means that it is possible for the `1` to be printed *after* `test-strict2` has returned a `4`! On the other hand, data and control dependencies guarantee that both the `2` and `3` are printed before the multiplication in the body of `test-strict2`.

Eagons are useful for controlling fine-grained operational aspects of programs. For example, the `fork2` operator introduced in Section 4.2.5 can be expressed in terms of eagons:

```
(fork2 E1 E2)
  desugars to
(let ((x (eagon E1))
      (y (eagon E2)))
  #t)
```

Eagons are used extensively in the implementation of SYNAPSE to manage the details of demand-driven evaluation.

Eagons can also be used to define an eager argument evaluation strategy. Consider the `ecall` special form, defined by the following desugaring:

```
(ecall Eproc Earg1 ... Eargn)
  desugars to
(pcall (eagon Eproc)
       (eagon Earg1)
             ⋮
       (eagon Eargn)
       )
```

The `pcall` and `eagon`s allow all subexpressions of the `ecall` to be evaluated in parallel. But the `eagon`s on the arguments further allow the arguments to be evaluate concurrently with the body of the procedure. `Ecall` corresponds to the default procedure application strategy for the Id programming language [AN89, Tra88].

7.1.6 Graphical Bindings

OPERA's `nex` and `nexrec` constructs are "graphical" versions of `let` and `letrec`. Whereas `let` and `letrec` associate names with first-class values, `nex` and `nexrec` associate names with syntactic entities so that the general graph-structured syntactic dependencies can be expressed within the tree-structured confines of s-expressions. `Nex` and `nexrec` are essentially lazy forms of `let` and `letrec` embedded within a strict language.

We motivate these constructs and explain their semantics via a simple example. Consider the following OPERA procedure:

```
(lambda (a b x y)
  (cons (if (test a)
            (compute x)
            (compute y))
        (if (test b)
            (compute x)
            (compute y))))
```

Suppose that `test` is a unary predicate and `compute` is an expensive unary procedure with no side effects. In some cases, the procedure performs more work than necessary: `(compute x)` is evaluated twice if both tests are true, and `(compute y)` is evaluated twice if both tests are false.

One way to avoid the extra computation is to employ a finer-grained case analysis:

```
(lambda (a b x y)
  (let ((ta (test a))
        (tb (test b)))
    (if ta
        (if tb
            (let ((cx (compute x))) (cons cx cx))
            (cons (compute x) (compute y)))
        (if tb
            (cons (compute y) (compute x)))
            (let ((cy (compute y))) (cons cy cy)))))
```

Unfortunately, this approach significantly impairs readability.

Another approach is to use the laziness and memoization provided by lazons to achieve the appropriate effect:

```
(lambda (a b x y)
  (let ((cx (lazon (compute x)))
        (cy (lazon (compute y))))
    (cons (if (test a) (touch cx) (touch cy))
          (if (test a) (touch cx) (touch cy)))))
```

Explicit touches are required in this case because the operands of `cons` constructor are not implicitly touched. This approach leads to a more readable program than the finer-grained case analysis, but the extra `lazon`s and `touch`es are distracting management details.

Figure 7.3 uses the graphical notation introduced earlier (and formalized in Chapter 8) to depict the desired structure for the body of the sample procedure. The results of both `compute` applications are shared by both `if`s. According to the demand-driven model detailed in Chapter 8), each call to `compute` is evaluated only once if the result is required, and is not evaluated at all if the result is not required.

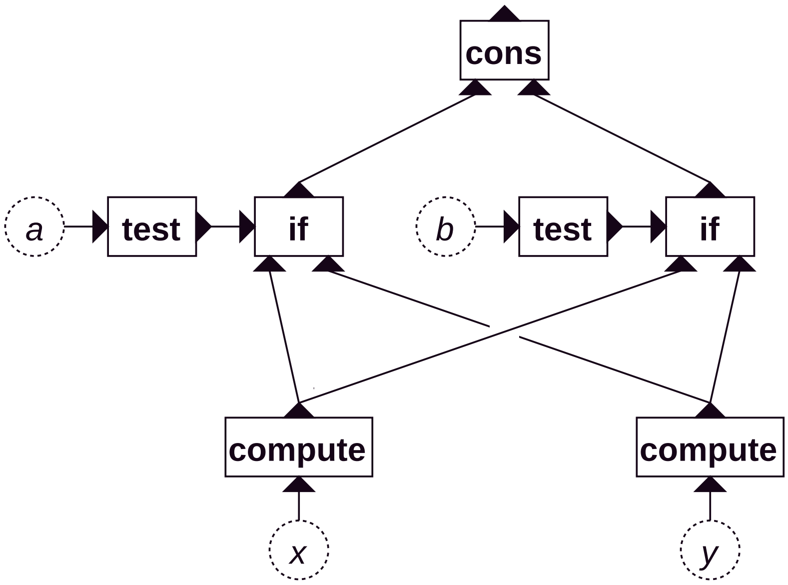


Figure 7.3: Graphical depiction of a program whose sharing properties are difficult to express in textual syntax.

The `nex` binding construct expresses the sharing exhibited by the graphical notation within textual OPERA code. Here is a version of the sample procedure that uses `nex`:

```
(lambda (a b x y)
  (nex ((cx (compute x))
        (cy (compute y)))
    (cons (if (test a) cx cy)
          (if (test a) cx cy))))
```

Intuitively, `nex` associates the names `cx` and `cy` with the outputs of the graph structure corresponding to the `compute` applications, and any references to these names in the body of the `nex` "wire" the appropriate output into the graph structure corresponding to the body. For this reason, we refer to `nex` as a graphical binding. The meaning of the `nex` version is actually defined in terms of the graph model, but it can also be viewed as an more convenient syntax for the explicit `lazon`/`touch` approach (i.e., we could desugar any `nex` expression into an expression using explict `lazon`s and `touch`es).

In general, `nex` behaves differently from `let`. If `nex` were replaced by `let` in the above example, the strictness of OPERA would dictate that both `(compute x)` and `(compute y)` would be evaluated before the body, regardless of whether both their values were used.

`Nexrec` is similar to `nex`, but has the recursive scoping of a `letrec`. For example, the OPERA expression

```
(nexrec ((a (cons 1 b))
         (b (cons 2 a)))
  (cons a b))
```

corresponds to the graphical syntax depicted in Figure 7.4.[7]

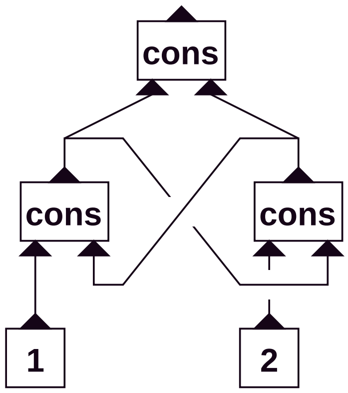


Figure 7.4: Graphical syntax exhibiting mutual dependencies that can be expressed with `nexrec`.

7.2 Implementing SYNAPSE

The OPERA language is equipped with the concurrency, synchronization, and non-strictness machinery necessary to implement slivers and slags. For the sake of concreteness, we will illustrate OPERA implementations of some the SYNAPSE slivers introduced in Chapter 6.

Implementing the elements of the lock step processing model in OPERA is a challenging activity. The features provided by OPERA for controlling the operational behavior of programs are very powerful but also rather low-level. The dizzying number of ways in which these features can be combined to yield slightly different computations can easily lead to mental overload. Yet, in order to achieve the goal of operational faithfulness, an imple-

[7]This example is somewhat contrived. According to the rules of the EDGAR model introduced in Chapter 8, evaluating the sample `nexrec` expression leads to deadlock. It is necessary to introduce some non-strictness to avoid deadlock.

menter of slivers must meticulously consider the consequences of every design choice. Here are some sample questions that continually confront the sliver implementer:

- How do I propagate demand to this subexpression without demanding these other subexpressions?
- Does this sliver release its hold on storage resources as soon as it can?
- Should I express this timing constraint with sequencing, an explicit `wait`, or a blocking reference to a synchron?
- How can I ensure that this timing constraint doesn't introduce a spurious deadlock?
- Should this expression be left as is, or does it need to be wrapped in a `lazon` or `eagon`?
- Do I need an explicit `touch` here or not?

Because the complexities of sliver implementation are so overwhelming, it is unreasonable to expect that average programmers should ever have to deal with them. Instead, programmers should be provided with easy-to-use abstractions that hide all the messy details. The SYNAPSE procedures of Chapter 6 are in this spirit. The kinds of questions listed above need only be considered by the expert programmers implementing the abstractions.

In order to simplify the presentation, I will discuss the details of sliver implementation in stages. First, I introduce some conventions and abstractions for manipulating slags that simplify the subsequent discussion. Then I describe how to implement a few linear slivers ignoring the issues of filtering. Next I show how to properly handle filtering for the linear slivers. Finally, I conclude with some notes on the implementation of tree-shaped slivers.

7.2.1 Slag Conventions

Representing and manipulating slags is a tricky business for several reasons:

- Slags contain references to synchrons. If these references aren't handled carefully, spurious deadlocks will result.
- The laziness aspect of slags means that components of a slag should not be computed until they are required. Including just the right amount of laziness can be challenging.

- The aggregate nature of slags means that they can potentially consume large amounts of storage. When manipulating slags, it is necessary to exercise great care to ensure that there is no unnecessary consumption of storage resources.

Here we present a set of conventions and some abstractions that simplify the manipulation of slags.

Recall that a slag is an aggregate whose skeletal nodes are wired together with synters (synchronized pointers). We will represent a synter as an entity consisting of two synchrons (one down, one up) and a lazon that computes the skeletal node pointed at by the synter. Figure 7.5 presents a stylized schematic of a synter. The nodes labelled **S** are synchrons and the node labelled **L** is a lazon.

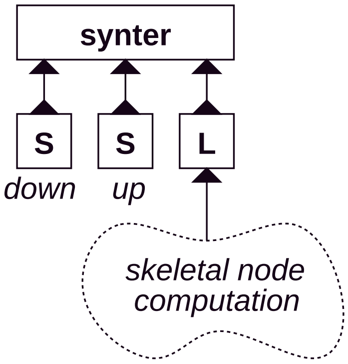


Figure 7.5: Graphical depiction of the structure of a synter.

A synquence (synchronized lazy list) is a slag whose skeletal nodes are either a distinguished end-of-list marker or a pair of a lazily computed element and child synquence (see Figure 7.6). A syndrite (synchronized lazy tree) is a slag whose skeletal nodes are pairs of a lazily computed element and a list of children syndrites (see Figure 7.7).

The basic structure illustrated in the figures is augmented with timing constraints that define when the computations suspended by the lazons actually take place. For a synter, the computation of the skeletal node takes place after the rendezvous at the down synchron. Attempting to manipulate the skeletal node before this point in time will result in deadlock.

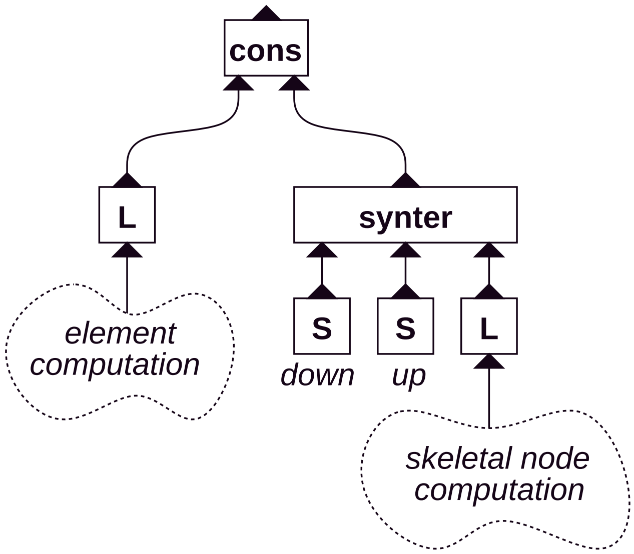


Figure 7.6: Structure of the skeletal node of a synquence.

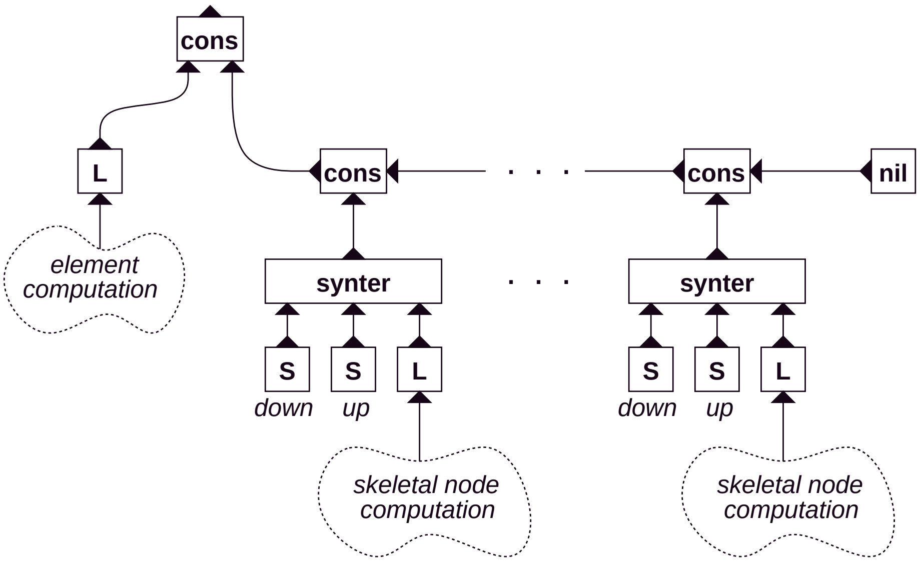


Figure 7.7: Structure of the skeletal node of a syndrite.

The timing of the element computation of a synquence depends on the shapes of the slivers producing and consuming it. If the producing sliver has *down* shape, then the element must be computed before the time of the down synchron of the child synquence. If the producing sliver has *up* shape, then the element must be computed after the rendezvous of the up synchron of the child synquence but before the rendezvous of the up synchron of the synquence itself. If the producing synquence has *across* shape, then the timing constraints are determined by context (see Section 5.2.3).

The timing constraints on the computation of a syndrite element are even more varied. They express when the element is computed relative to the down and up synchrons of the given syndrite and its children syndrites. It is a dynamic error for a program to require the value of a synquence or syndrite element at a time that precedes the time at which it is actually computed.

Taken together with the sliver requirements of Section 5.4.2, the structure depicted in Figures 7.5 – 7.7 and the timing constraints sketched above comprise a set of conventions for implementing slivers. On the one hand, the conventions determine the properties of an input slag that a sliver implementer can depend on. For example, an implementer knows that that all the synters for the children of a syndrite will be available before a rendezvous on any child's down synchron. On the other hand, the conventions limit the ways in which the implementer may produce output slags. As the flip side of the previous syndrite example, an implementer must guarantee that all the synters for the children of a produced syndrite must be available before the computation of any of the skeletal nodes of those children.

7.2.2 Slag Abstractions

Carefully following all the conventions outlined above can be difficult because there are so many details to monitor. Here we present a few OPERA procedures that facilitate slag manipulation by packaging up some of the conventions into handy abstractions.

Synters will be represented as three-element lists assembled via `synter` and disassembled via `unsynter`:

```
(define (synter down up $snode)
  (list down up $snode))

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

The `down` and `up` arguments of `synter` should be synchrons, and the `$snode` argument should be a lazon that computes the skeletal node. The `$` in `$snode` is a naming convention indicating that the variable names a lazon.

The following abstractions capture the notion of a rendezvous occuring at a procedure call boundary:[8]

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

(define (call/down-up down up proc . args)
  (seqn (wait down)
        (seq1 (apply proc args)
              (wait up))))
```

`Call/down` uses the synchron `wait` procedure to rendezvous with other slivers at a `down` synchron before a tail call of `proc` applied to `args`. `Call/down-up` is similar, but also engages in a rendezvous at the `up` synchron upon return from `proc` to model a non-tail call.

Figure 7.8 presents some abstractions that are helpful for satisfying the rendezvous requirements for procedure calls that manipulate slags. `Unslag/down` and `unslag/down-up` are variants of `call/down` and `call/down-up` that specially handle the case where the first argument is a slag. To allow demand to propagate through a chain of nested sliver applications, it is necessary in this case to eagerly evaluate the (otherwise delayed) computation that produces the skeletal node of the slag. Every skeletal node computation must perform a `wait` on the down synchron before doing any real work. If this computation were not eagerly evaluated, then the `wait` performed by `call/down` or `call/down-up` would certainly deadlock. The `wait`s within the skeletal node computations guarantee that the eager evaluation of one skeletal node can't race too far ahead of another.

`Reslag/down`, `reslag2/down` and `reslag/down-up` are useful for implementing slivers that map one or more input slags to an output slag. `Reslag/down` maps an input slag to

[8] Since the current implementation of OPERA does not support rest arguments, these procedures and other procedures using rest arguments are actually implemented as macros.

```
(define (unslag/down proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down down
        proc (eagon (touch $snode))
             args))))

(define (unslag/down-up proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down-up down up
        proc (eagon (touch $snode))
             args))))

(define (reslag/down proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (synter down up
        (lazon (apply call/down down
                 proc (eagon (touch $snode))
                      args))))))

(define (reslag2/down proc slag1 slag2 . args)
  (unsynter slag1
    (lambda (down1 up1 $snode1)
      (unsynter slag2
        (lambda (down2 up2 $snode2)
          (let ((down (simul! down1 down2))
                (up (simul! up1 up2)))
            (synter down up
              (lazon (apply call/down down
                       proc (eagon (touch $snode1))
                            (eagon (touch $snode2))
                            args)))))))))

(define (reslag/down-up proc slag receiver . args)
  (unsynter slag
    (lambda (down up $snode)
      (nex ((new-snode&other (apply call/down down
                               proc (eagon (touch $snode))
                                    args)))
        (nex ((new-snode (car new-snode&other))
              (other (eagon (seq1 (touch (cdr new-snode&other))
                                  (wait up)))))
          (let (($other (lazon other)))
            (receiver (synter down up (lazon (seq1 new-snode other)))
                      $other)))))))
```

Figure 7.8: Abstractions for procedure application that facilitate the implementation of rendezvous requirements.

an output slag in which every skeletal node of the output is determined by applying `proc` to the corresponding skeletal node of the input. It is essentially a slag-returning version of `unslag/down`. `Reslag2/down` is a similar procedure that maps two input slags to an output slag. In order to handle fan-in, `reslag2/down` uses `simul!` to unify the corresponding synchrons of its two inputs.

`Reslag/down-up` is a slag-returning version of `unslag/down-up`. Its interface is complicated by the fact that it returns not only a new slag, but also an accumulator (`other`) that is computed in the *up* phase. The `receiver` argument to `reslag/down-up` is a procedure that accepts these two results. The `proc` argument, which is applied to the input slag and the other arguments, is expected to return a pair of the new slag and up accumulator. The intricacies in the definition of `reslag/down-up` are due the complex collection of timing constraints that must be satisfied. In particular, the new slag must be available in the *down* phase of the recursion, but the up accumulator will not be computed until the *up* phase of the recursion. To get this effect, the up accumulators returned by `proc` and returned to `receiver` must be appropriately delayed. The definition uses `nex`, `eagon`, and `seq1` to implement intricate mechanisms for ensuring that operations occur in an order that avoids spurious deadlocks and space build-up.

While the procedures presented in this section are helpful in many cases, they aren't appropriate for all situations. Implementing slivers sometimes requires manipulating low-level details in a way that is incompatible with the abstractions developed here.

7.2.3 Unfiltered Synquences

As an introduction to sliver implementation, we first consider *unfiltered synquences* — i.e., synquences whose elements are all present. This permits us to study the essence of the synchronization issues without the considerable complications associated with filtering. We will describe the implementations of a few representative slivers: the `genQ` generator, the `map2Q` mapper, the `downQ` and `upQ` accumulators, and the `down-scanQ` and `up-scanQ` scanners.

GenQ

`GenQ` creates a synquence from scratch:

```
(define (genQ init next done?)
  (define (gen-synq $arg up-parent)
    (let ((down (synchron))
          (up (synchron)))
      (precede! up up-parent)
      (synter down up
        (lazon (call/down down gen-list (touch $arg) up)))))
  (define (gen-list arg up)
    (if (done? arg)
        '()
        (cons (lazon arg) (gen-synq (lazon (next arg)) up))))
  (gen-synq (lazon init) (synchron)))
```

The internal `gen-synq` procedure creates two fresh synchrons and packages them up into a synter, while the `gen-list` procedure returns a skeletal pair node for the new synquence. The `up-parent` argument to `gen-synq` is the up synchron of previous call; it is passed in to permit the declaration that a rendezvous at the current up synchron must precede a rendezvous at its parent. The `(synchron)` in the initial call to `gen-synq` is an arbitrary synchron that serves as the parent to the first up synchron in the resuling synquence. The `$arg` argument to `gen-synq` is delayed so that it will only be computed if `done?` tests false.

In the above code, the `lazon`s in the `(lazon arg)` and `(lazon init)` expressions are actually superfluous. Since `arg` and `init` are already guaranteed to be values, there is no need to delay their computation. Henceforth, we will eliminate superfluous lazons without comment.

Map2Q

`Map2Q` uses `reslag2/down` to map two input synquences into one output synquence:

```
(define (map2Q fun synq1 synq2)
  (reslag2/down synq1 synq2
    (lambda (lst1 lst2)
      (if (or (null? lst1) (null? lst2))
          '()
          (let (($elt1 (car lst1))  ; Aggressive unbundling
                ($elt2 (car lst2))) ;
            (cons (lazon (fun $elt1 $elt2))
                  (map2Q fun (cdr lst1) (cdr lst2)))))))
```

The handling of the `null?` tests guarantees that the length of the output synquence is the shorter of the two input synquences. Truncation slivers can be handled in a similar fashion.

Using `let` to name the `car`s of `lst1` and `lst2` is a crucial application of the aggressive unbundling principle set forth in Section 5.4.2. If the expression

```
(let (($elt1 (car lst1))
      ($elt2 (car lst2)))
  (cons (lazon (fun $elt1 $elt2))
        (map2Q fun (cdr lst1) (cdr lst2))))
```

were instead replaced by

```
(cons (lazon (fun (car lst1) (car lst2)))
      (map2Q fun (cdr lst1) (cdr lst2)))
```

then the `map2Q` sliver could lead to deadlock in certain contexts. The reason is that `lst1` and `lst2` refer to synters in their `cdr` slots, and those synters contain references to down synchrons. Even though the element computation would not require these synters, the references to them would not be dropped until the element computation was demanded. If the element computation were not demanded until the *up* phase, then the rendezvous on the down synchrons would be blocked and deadlock would ensue. This example is typical of the kind of careful reasoning that must be performed in the presence of synchrons to avoid deadlock.

DownQ and UpQ

Except for the use of the synchronized calling abstractions, the down and up accumulators look like classical list accumulators:

```
(define (downQ init op synq)
  (define (accum lst acc)
    (if (null? lst)
        acc
        (unslag/down accum (cdr lst) (op (car lst) acc))))
  (unslag/down accum synq init))

(define (upQ init op synq)
  ;; OP's first arg is lazy and second arg is eager
  (define (accum lst)
    (if (null? lst)
        init
        (op (car lst)
            (eagon (unslag/down-up accum (cdr lst))))))
  (unslag/down-up accum synq))
```

DownQ uses unslag/down to preserve tail recursion, while upQ uses unslag/down-up to return from every call.[9]

The careful reader may be curious about the presence of an eagon within upQ. The justification for this eagon is rather subtle. We shall motivate its purpose in depth as an example of the kinds of details and design choices that makes sliver implementation so challenging.

The eagon within the definition upQ is intended to enhance the operational faithfulness of that sliver. Without the eagon, upQ can unduly restrict the timing of of element computations. Consider the SYNAPSE expression:

```
(upQ (lambda ($elt acc) (+ $elt acc))
     0
     (mapQ square (to-1 3)))
```

Since the square mapper is arranged between a down and an up sliver, we expect that the relative order of squares should be unconstrained (see Section 5.2.3). But an eagon-less implementation of upQ will force all the squares to happen in the *up* phase of the computation! Why? Since synquence elements are lazy, they are not computed until their value is required. By inspecting the body of the lambda expression, *we* can tell that all of the results of the square applications *will* be needed — as arguments to +. But the OPERA interpreter doesn't look inside the body of the lambda until it applies the procedure, and it only does that when both arguments are available. Since the second argument does not become available until the *up* phase, none of the square applications will be demanded until the *up* phase.

Inserting an eagon around the second argument allows (but does not require) the application of op to occur in the *down* phase. If the body of op requires the element denoted by its first argument (as it does in the above example), the element computation may be performed in the *down* phase. If the body of op does not require the value of this element,

[9] There really should be two versions of upQ: one in which the initial call is made via unslag/down, and another in which it is made via unslag/down-up. The former would be used in situations where upQ appears in a tail call position, and the latter would be used for calls in non-tail positions. Similarly, there should be two versions of downQ. We have simplified matters here by considering only the most common case. The need for different versions underscores the fact that the slivers are explicitly *simulating* tail calls. It would be preferable to have a system that automatically inserted an up rendevous only when it was necessary. However, preliminary attempts at such a system have been unsuccessful.

it will not be computed at all. This behavior better matches the expected behavior of `upQ`.

Unfortunately, the fix changes the interface to `op` in a subtle way. The fact that the second argument to `op` is now an eagon can lead to unexpected behavior for an `op` that does not require the value of its second argument. Consider the following SYNAPSE expression:

```
(upQ (lambda ($elt acc) (display $elt))
     'ignore
     (to-1 3))
```

Naively, we might expect this expression to print out numbers during the *up* phase in the order `1`, `2`, `3`. And this *would* be the behavior in the `eagon`-less version of `upQ`. But in the `eagon` version of `upQ`, nothing prevents the numbers to print out during the *down* phase in the order `3`, `2`, `1`. In such cases, getting the desired behavior may require extending `op` with an explicit control dependency:

```
(upQ (lambda ($elt acc) (seqn (touch acc) (display $elt))
     'ignore
     (to-1 3))
```

In this case, requiring the `touch` of `acc` to occur before the `display` expresses the constraint that all the `display`s should occur in reverse order during the *up* phase.

Down-scanQ and Up-scanQ

`Down-scanQ` and `up-scanQ` are scanning transducers patterned after the `downQ` and `upQ` accumulators. The definition of `down-scanQ` is essentially a version of `downQ` that uses `reslag/down` in place of `unslag/down`:

```
(define (down-scanQ init op synq)
  (define (scan lst acc)
    (if (null? lst)
        '()
        (let ((new-acc (op (car lst) acc)))
          (cons new-acc
                (reslag/down scan (cdr lst) new-acc)))))
  (reslag/down scan synq init))
```

The internal `scan` procedure maps a given skeletal node (`lst`) into one that holds the new value of a down accumulator.

Although the definition of `up-scanQ` is also closely related to that of `upQ`, the resemblance is clouded by the more complex interface of `reslag/down-up`:

```
(define (up-scanQ init op synq)
  ;; OP's first arg is lazy and second arg is eager
  (define (scan lst)
    (if (null? lst)
        (cons '() init)
        (let (($elt (car lst)))
          (reslag/down-up scan (cdr lst)
            (lambda (new-list $sub-acc)
              (let (($up-acc (lazon (op $elt $sub-acc))))
                (cons (cons $up-acc new-list)
                      $up-acc)))))))
  (reslag/down-up scan synq (lambda (final-synq $final-acc) final-synq)))
```

In this case, `scan` maps a given skeletal node (`lst`) into a pair of (1) a new skeletal node and (2) a lazon of the calculation of the up accumulator. The `lazon` is necessary because the pair must be returned in the *down* phase, long before the application of `op` will take place in the *up* phase. Removing the `lazon` would lead to a deadlock because then the pair could not returned until the *up* phase even though its `car` would be required by the *down* phase.

The delayed up accumulator, `$up-acc`, is used both by the new skeletal node and the `cdr` of the returned pair. Since the accumulator is already present in the returned skeletal node, why is it also necessary to return it as a separate element of the pair? The answer is that it helps to make the base case work smoothly. Since the empty list does not contain the initial accumulator `init`, an alternate mechanism must be found for associating `init` with the empty list. Returning the accumulator in the second slot of the pair avoids having to test `new-list` with `null?` upon every return. If `up-scanQ` were modified so that it did not take an initial accumulator value but instead started accumulation with the last element of the input, this workaround would be unnecessary.

7.2.4 Filtered Synquences

As described in Section 5.5.3, it is tricky to design reusable slivers that deal with filtering in a reasonable way. Here, we present an approach to synquence filtering that addressed the issues raised in that section. Fortunately, we will be able to use many of the same synchronization abstractions that proved handy for the unfiltered case. Indeed, by hiding unnecessary detail, those abstractions help to highlight what is different about the filtered case.

New Abstractions

We begin by defining some new abstractions designed specifically for filtering. In the unfiltered case, the element held by a skeletal node is represented as a lazy computation. In the filtered case, each element is annotated with two flags:

1. The *down-element?* flag indicates whether the associated element is computed in the down or up phase.
2. The *down-gap?* flag indicates whether the element slot holds a gap in the *down* phase of a computation. If *down-gap?* is true, the associated element must be a gap. If *down-gap?* is false, then the associated element might be a non-gap value determined in the *down* phase, or it might be any value (including a gap) determined in the *up* phase.

All four combinations of flags are possible. The most "interesting" combination is when the *down-element?* is false but *down-gap?* is true. This means that a slot produced by a *up* sliver is known to contain a gap during the *down* phase even though the ungapped slots produced by the sliver are all computed in the *up* phase. This explicit shape information makes general filtering mechanisms possible by informing slivers when it's safe to test a slot for a gap.

An element and its two flags will be bundled together into a structure called a *filton*. These structures are built by `filton` and taken apart by `unfilton`:

```
(define (filton down-elt? down-gap? $elt)
  (list down-elt? down-gap? $elt))

(define (unfilton fil receiver)
  (receiver (car fil) (cadr fil) (caddr fil)))
```

It is also helpful to have functions that map between filtons. Given a filton, `refilton` returns a new filton with the same flags in which a given function is lazily applied to the original element:

```
(define (refilton fil f)
  (unfilton fil
    (lambda (down-elt? down-gap? $elt)
      (filton down-elt? down-gap?
        (lazon (let ((e (eagon (touch $elt))))
                 (if (gap? e)
                     #g      ; The gap literal
                     (f e))))))))
```

We assume that `f` never returns a gap as a result, because that might violate the validity of the `down-gap?` flag. `Refilton2` is similar, but takes two inputs:

```
(define (refilton2 fil1 fil2 f)
  (unfilton fil1
    (lambda (down-elt1? down-gap1? $elt1)
      (unfilton fil2
        (lambda (down-elt2? down-gap2? $elt2)
          (filton (and down-elt1? down-elt2?)
                 (or down-gap1? down-gap2?)
                 (lazon (let ((e1 (eagon (touch $elt1)))
                              (e2 (eagon (touch $elt2))))
                          (if (or (gap? e1) (gap? e2))
                              #g   ; The gap literal
                              (f e1 e2))))))))))
```

The resulting filton holds a gap if either input holds a gap. The *down-gap?* flag of the result is true if it is true of either input, but the resulting element is produced in the down phase only if both inputs are produced in the *down* phase.

The key constraint to be obeyed in filton manipulation is that the two flags of a filton must be accessible in the *down* phase of a computation. The element itself may be computed at any time, but its enclosing filton must contain a lazon of this computation in the *down* phase.

Implementing Filtered Slivers

We study filtering by examining some sliver implementations. First we consider the `filterQ` sliver, and then we reconsider the same six slivers studied for the unfiltered case.

`FilterQ` is the only primitive sliver capable of introducing gaps where they did not exist before. An implementation of the synquence filtering sliver, `filterQ`, is shown in Figure 7.9. `FilterQ` is a special kind of mapping sliver that maps an element either to itself or to a gap. As in most slivers on filtered synquences, the action taken by `filterQ` depends on the values of the filton flags. If the element is known to be a gap in the *down* phase the filton is passed untouched. Otherwise the action depends on when the element will be available. If the element is available in the *down* phase (i.e., *down-element?* is true), then it can be immediately tested with `pred`. But if the element won't be available until later (i.e., *down-element?* is false), then a `gap?` test and `pred` test must be delayed until the *up* phase. In all cases, the sliver must produce a valid filton object in the *down* phase so that other slivers can examine it during the *down* phase.

```
(define (filterQ pred synq)
  (define (filter-list lst)
    (if (null? lst)
        '()
        (let ((fil (car lst)))
          (cons (unfilton fil
                   (lambda (down-elt? down-gap? $elt)
                     (if down-gap?
                         fil
                         (if down-elt?
                             (if (pred (touch $elt))
                                 fil
                                 (filton #t #t #g))
                             (filton #f #f
                               (lazon (let ((elt (touch $elt)))
                                        (if (gap? elt)
                                            #g
                                            (if (pred elt) elt #g)))))))))
                (reslag/down filter-list (cdr lst))))))
  (reslag/down filter-list synq))
```

Figure 7.9: An implementation of the `filterQ` sliver for filtering synquences.

Filtered versions of `genQ` and `map2Q` appear in Figure 7.10. These are minor tweaks to the unfiltered versions; only the lines marked with `;***` have been changed. Every filton created by `genQ` contains ungapped elements produced in *down* phase (i.e., *down-element?* is true, *down-gap?* is false). `Map2Q` calls `refilton2` to do the actual mapping work.

Filtered versions of `downQ` and `down-scanQ` (Figure 7.11) differ from their unfiltered counterparts by treating gaps specially. Each avoids performing an accumulation step in the case where *down-gap?* is true.

Extending the unfiltered versions of `UpQ` and `up-scanQ` to filtered versions requires more substantial changes (Figure 7.12). When *down-gap?* is true, `upQ` can effectively make a

```
(define (genQ init next done?)
  (define (gen-synq $arg prev-up)
    (let ((down (synchron))
          (up (synchron)))
      (precede! up prev-up)
      (synter down up
        (lazon (call/down down gen-list (touch $arg) up)))))
  (define (gen-list arg up)
    (if (done? arg)
        '()
        (cons (filton #t #f arg)                  ; ***
              (gen-synq (lazon (next arg)) up))))
  (gen-synq init (synchron)))


(define (map2Q fun synq1 synq2)
  (define (map2-list lst1 lst2)
    (if (or (null? lst1) (null? lst2))
        '()
        (let ((fil1 (car lst1))
              (fil2 (car lst2)))
          (cons (refilton2 fil1 fil2 fun)         ; ***
                (reslag2/down map2-list (cdr lst1) (cdr lst2))))))
  (reslag2/down map2-list synq1 synq2))
```

Figure 7.10: Filtered implementations of `genQ` and `map2Q`.

```
(define (downQ init op synq)
  (define (accum lst acc)
    (if (null? lst)
        acc
        (unfilton (car lst)
          (lambda (down-elt? down-gap? $elt)
            (if down-elt?
                (if down-gap?
                    (unslag/down accum (cdr lst) acc)
                    (unslag/down accum (cdr lst) (op $elt acc)))
                (error "DOWNQ: Shape error"))))))
  (unslag/down accum synq init))

(define (down-scanQ init op synq)
  (define (down-list lst acc)
    (if (null? lst)
        '()
        (unfilton (car lst)
          (lambda (down-elt? down-gap? $elt)
            (if down-elt?
                (if down-gap?
                    (cons (filton #t #f acc)
                          (reslag/down down-list (cdr lst) acc))
                    (let ((new-acc (op $elt acc)))
                      (cons (filton #t #f new-acc)
                            (reslag/down down-list (cdr lst) new-acc))))
                (error "DOWN-SCANQ: Shape error"))))))
  (reslag/down down-list synq init))
```

Figure 7.11: Filtered implementations of `downQ` and `down-scanQ`.

tail call; but if *down-gap?* is false `upQ` must defer an explicit `gap?` test until the *up* phase. `Up-scanQ` is similar, but is complicated by the need to create a valid filton in the *down* phase even though all the applications of `op` will occur in the *up* phase.

Expressing timing constraints for *up* slivers proves to be quite tricky in the filtered case. In `upQ` and `up-scanQ`, the goal is to require each computation of `op` to occur between a consecutive pair of up synchrons. But the slag manipulation abstractions developed above only rendezvous on the up synchron corresponding to the input of `op`; nothing explicitly forces `op` to return before the up synchron corresponding to its output. This does not cause a problem in the unfiltered case, because every position of the recursion is occupied with an instance of `op`. But in the filtered case, some positions are not occupied, and the output of an `op` may not be properly constrained.

To fix this problem, it is necessary to extend the slag abstractions to pass down an up synchron from above. Figure 7.13 shows new versions of these abstractions in which each `proc` takes an additional argument, called a *returner*. A returner is a unary procedure that maintains a non-waiting pointer to an up synchron. The procedure acts as the identity, but calling the procedure also releases the synchron pointer, which in turn may enable a rendezvous at the synchron. Both `upQ` and `up-scanQ` use the modified slag abstractions to obtain access to a returner (bound to the variable `return`). The return is wrapped around non-trivial up computation to guarantee that it is bounded from above as well as from below.

7.2.5 Syndrites

The implementation of syndrites follows the same ideas used in the implementation of synquences. In fact, the same slag abstractions that aid the synquence implementations are useful for many of the syndrite implementations. However, the fact that a syndrite skeletal node generally has multiple children can introduce many headaches.

To give the flavor of syndrite slivers, I will briefly describe the implementations of three simple unfiltered examples. Figure 7.14 shows the implementations of `mapD`, `preD`, and `down-scanD`.

The `mapD` syndrite mapper closely resembles synquence mappers. The main difference

```
(define (upQ init op synq)
  (define (accum lst return)
    (if (null? lst)
        (return init)
        (unfilton (car lst)
          (lambda (down-elt? down-gap? $elt)
            (if down-gap?
                (unslag/down-return accum (cdr lst))
                (let ((sub-acc (eagon (unslag/down-up-return accum (cdr lst)))))
                  (return
                    (if (gap? $elt)
                        (touch sub-acc)
                        (op $elt sub-acc))))))))))
  (unslag/down-up-return accum synq))

(define (up-scanQ init op synq)
  (define (up-list lst return)
    ;; Returns a pair of (1) A new list (2) A (lazy) new accumulator
    (if (null? lst)
        (cons '() (lazon (return init)))
        (let ((fil (car lst))
              (subsynq (cdr lst)))
          (unfilton (car lst)
            (lambda (down-elt? down-gap? $elt)
              (if down-gap?
                  ;; Return a non-gap in either case.
                  (reslag/down-return-receiver up-list subsynq
                    (lambda (new-list $sub-acc)
                      (cons (cons (filton #f #f $sub-acc)
                                  new-list)
                            $sub-acc)))
                  (reslag/down-up-return up-list subsynq
                    (lambda (new-list $sub-acc)
                      ;; This is tricky.  In all cases, MUST return a pair of
                      ;; a synchron and a lazy accumulator immediately.
                      (nex ((up-acc (if (gap? $elt) $sub-acc (op $elt $sub-acc))))
                        (cons (cons (filton #f #f (lazon up-acc))
                                    new-list)
                              (lazon (return up-acc))))))))))))
  (reslag/down-up-return up-list synq (lambda (final-synq $final-acc) final-synq)))
```

Figure 7.12: Filtered implementations of `upQ` and `up-scanQ`.

```
(define (unslag/down-return proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down down
        proc (eagon (touch $snode))
                    (lambda (x) (seq1 x up)) ; Returner
                    args))))

(define (unslag/down-up-return proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down-up down up
             proc (eagon (touch $snode))
                  (lambda (x) (seq1 x up)) ; Returner
                  args))))

(define (reslag/down-return-receiver proc slag receiver . args)
  (unsynter slag
    (lambda (down up $snode)
      (nex ((new-snode&other (apply call/down down
                               proc (eagon (touch $snode))
                                    (lambda (x) (seq1 x up)) ; Returner
                                    args)))
        (nex ((new-snode (car new-snode&other))
              (other (eagon (touch (cdr new-snode&other)))))
          (let (($other (lazon other)))
            (receiver (synter down up (lazon (seq1 new-snode other)))
                      $other)))))))

(define (reslag/down-up-return proc slag receiver . args)
  (unsynter slag
    (lambda (down up $snode)
      (nex ((new-snode&other (apply call/down down
                               proc (eagon (touch $snode))
                                    (lambda (x) (seq1 x up)) ; Returner
                                    args)))
        (nex ((new-snode (car new-snode&other))
              (other (eagon (seq1 (touch (cdr new-snode&other))
                                  (wait up)))))
          (let (($other (lazon other)))
            (receiver (synter down up (lazon (seq1 new-snode other)))
                      $other)))))))
```

Figure 7.13: Modified slag abstractions used by `upQ` and `up-scanQ`.

```
(define (mapD f synd)
  (define (map-tree tr)
    (let (($elt (elt tr)))
      (tree (lazon (f $elt))
            (map (lambda (kid) (mapD f kid))
                 (kids tr)))))
  (reslag/down map-tree synd))

(define (down-scanD init op synd)
  (define (accum tr acc)
    (let (($elt (elt tr))
          (subdrites (kids tr)))
      (let ((new-acc (op $elt acc)))
        (tree new-acc
              (map (lambda (synd) (reslag/down accum synd new-acc))
                   subdrites)))))
  (reslag/down accum synd init))

(define (preD dir init op synd)
  (define (walk-tree tr return acc)
    (let (($elt (elt tr))
          (ks (kids tr)))
      (let ((new-acc (op $elt acc)))
        (if (null? ks)
            (return new-acc)
            (walk-trees ((permuter dir) ks) new-acc)))))
  (define (walk-trees synds acc)
    (let ((fst (car synds))
          (rest (cdr synds)))
      (if (null? rest)
          (unslag/down-return walk-tree fst acc)
          (walk-trees rest (unslag/down-up-return walk-tree fst acc)))))
  (unslag/down-return walk-tree synd init))
```

Figure 7.14: Implementations of three unfiltered syndrites.

is that the internal `map-tree` manipulates a tree shaped skeletal node rather than a linear one. The `tree` constructor makes a tree out of an element and a list of subtrees; the `elt` and `kids` selectors extract these components from a tree. `MapD` is recursively mapped over the subtrees using OPERA's higher order `map` procedure. The base case that terminates the recursion is applying `map` to an empty list of children.

`Down-scanD` is a straightforward parallel down scanner that resembles the `down-scanQ` sliver for synquences.

Recall that `preD` is a pre-order accumulator that takes a direction argument (`dir`) in addition to the usual initial value, combiner, and slag. `PreD` is implemented as a pair of mutually recursive procedures, `walk-tree` and `walk-trees`. `Walk-tree` returns the accumulated value from a pre-order walk of a tree, while walk-trees returns the accumulated value from a pre-order walk of a list of syndrites. The `permuter` procedure used in the body of `walk-tree` permutes the children of a tree as specified by the direction.

The implementation of `preD` is carefully crafted to preserve important operational characteristics of a pre-order walk. To preserve tail recursion, the case where `walk-trees` is given a singleton list is handled specially; `walk-trees` is never called on an empty list. Returner-supplying versions of the slag abstractions are used to constrain the pre-order walking operations to happen at the right time with respect to other slivers that might be manipulating the same syndrite.

The simplicity of the above examples is somewhat deceiving. Many syndrite slivers, especially the scanners, are *very* complex and have repeatedly resisted simplification attempts. For example, the pre-order scanner is so intricate that it doesn't even fit on a single page! The scanners are particularly difficult to write because they must immediately return new syndrite structure while at the same time laying down the proper dependencies and time constraints for the accumulation computation that will take place later. Thus far I have been unable to find appropriate abstractions that elegantly express these intricate processes.

Another fly in the syndrite ointment is filtering. The two-flag synquence filtering technique does not solve the reusability problem for general syndrite filtering. In the synquence case, an element is filtered either in the *down* phase or the *up* phase. But in the syndrite case, filtering can happen *between* subcalls as well. This greatly complicates the design of filtered syndrites, and I have not worked out the details. The interactions between tree shapes and filtering remains an area for future study. The current implementation of SYNAPSE uses a straightforward approach to gap representation that is sufficient for the simple examples tested so far.

Chapter 8

EDGAR: Explicit Demand Graph Reduction

I have demonstrated how slivers and synchronized lazy aggregates can be realized using the concurrency, synchronization, and non-strictness features of OPERA. But since I have only described these features in an informal way, there are many potential ambiguities and unexplained subtleties concerning how they work. In particular, the details of how a rendezvous occurs at a synchron are far from clear.

In this chapter, I present a semantic model that precisely specifies the meaning of OPERA expressions. The model is based on *E*xplicit *D*emand *G*r*A*aph *R*eduction (EDGAR), a framework that I have developed for describing how computations unfold over time. Whereas many semantic frameworks focus on the *value* produced by a computation, EDGAR is designed to emphasize the *way* in which the value is computed. EDGAR represents a computation as a sequence of graphs, each one of which represents the control state and accessible data at a particular point in the computation. EDGAR shares many similarities with other models of computation (e.g., graph reduction, dataflow models, concurrency models, structured operational semantics). The key feature that distinguishes it from most other semantic models is its explicit representation of the flow of demand through a computation. This feature simplifies the explanation of OPERA's features for fine-grained operational control.

I present the semantics of OPERA in three parts. First, I explain the basics of the

EDGAR model. Then I focus on the EDGAR rules that implement OPERA's concurrency, synchronization, and non-strictness features. Finally, I describe how an OPERA expression can be evaluated by translating it into an initial EDGAR graph, turning the crank of the graph reduction process, and translating the final graph into an OPERA result. I conclude the chapter by relating EDGAR to other graph-based semantic frameworks.

8.1 The Basics of EDGAR

At the very highest level, EDGAR follows the basic recipe for an operational semantics framework [Plo81]. In an operational semantics, program execution states are represented by some sort of structured *configuration*, and there are *transition rules* that specify how the computation can step from one configuration to the next. Program execution is modelled by mapping the program to an *initial configuration* and iteratively applying transition rules until a suitable *final configuration* (one representing an answer) is obtained. The final configuration is then mapped to an appropriate answer domain.

8.1.1 Snapshots

In EDGAR, each configuration is called a *snapshot* because it represents a single moment in the dynamic evolution of a process. A snapshot is a graph consisting of interconnected program components. Figure 8.1 is a visual depiction of a sample snapshot that we will refer to in the ensuing discussion.

Each program component, called a *node*, can be viewed as a computational device that responds to a demand for a value by computing that value. Every node is has a *label* that indicates the behavior of the node. Each node posesses a set of labelled *input ports* that specify the arguments of the node and a set of labelled *output ports* that specify the results of the node. The number of input ports and output ports is dictated by the label of the node. Typically, a node has several input ports and one output port. Every node must have at least one port, but one of the input port set or the output port set may be empty.[1]

[1] In all of our examples, a node will have only 0 or 1 output ports, but in general it can have any number.

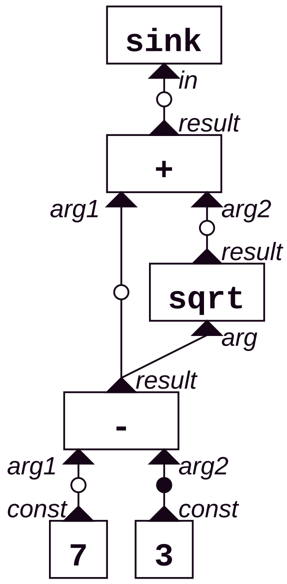


Figure 8.1: A simple snapshot.

In Figure 8.1, nodes are depicted as labelled rectangles, and ports are depicted as triangles attached to the nodes. An input port points into its node; an output port points away from its node. Ports are labelled with names that serve to distinguish them, as well as indicate their purpose.

Nodes are wired together much like the components of an electrical circuit. Each connection is indicated by a directed *wire* from an output port of the *source node* to an input port of the *target node*. Wires appear as lines in Figure 8.1. Technically, a wire connects a *source port* to a *target port*, but when there is no ambiguity it is convenient to refer to a wire as connecting two *nodes*. The set of wires entering the input ports of a node are its *input wires*, while those that leave its output ports are its *output wires*.

Intuitively, a wire is used for a two-step communication protocol between its source and target ports: the target port can request the value from its source port, and the source port can respond to the request by returning a value. Every wire has a *state* attribute that indicates the status of this communication protocol. There are three possible wire states:

1. *inactive*: No request has yet been made on the wire.

2. *requested*: A request has been made on the wire, but no value has been returned.

3. *returned*: A request has been made on the wire, and a value has been returned.

In Figure 8.1, an unannotated line is an inactive wire; a line with an empty circle is requested; and a line with a filled circle is returned. Wire states are the chief characteristic of EDGAR that distinguish it from traditional forms of graph reduction. In particular, the existence of a *requested* state is what makes EDGAR an "explicit demand" model.

The value returned by a returned wire is the source node of the wire. In EDGAR, only a subset of the nodes may appear at the source of a returned wire. These nodes are called *value nodes*. Examples of value nodes include constants, data structures, and procedures. The `+`, `-`, and `sqrt` nodes in Figure 8.1 examples of nodes that are not value nodes; they represent the applications of procedures but are not procedures themselves.

We shall see later that the state of a wire can change when a computation steps from one snapshot to another. As implied by the description of the wire states, the communication protocol on a wire over a sequence of snapshots is very limited:

- At most one request can ever be made on a wire.

- No value can be returned to a wire until a request has been made on that wire.

- At most one value can ever be returned to a wire.[2]

A wire, then, is a one-shot communication "fuse" that can be used for transmiting a single request and a single value before it is "used up". These restrictions on wires distinguish EDGAR from dataflow graph models, in which wires carry a stream of value tokens.

In a legal snapshot, every input port must be connected to exactly one wire, but an output port may be connected to any number of wires, including zero. Wire fan-out at an output port allows the result computed at one node to be shared as an input to other nodes. Wire sharing means that nodes can be arranged in directed acyclic graphs (DAGs) or even exhibit cycles (see Figure 7.4 on page 323 for an example of a cyclic graph).

[2]This is true only for the functional subset of EDGAR. In the presence of nodes that model side effects, the value represented by a wire may change over time. See [Tur94] for details.

The snapshot depicted in Figure 8.1 is an intermediate configuration in a simple numerical computation. The computation specifies the calculation of

$$(7-3)+\sqrt{(7-3)}$$

where the sharing of the `-` node indicates that the difference between 7 and 3 is to be calculated only once. The nodes labelled `7` and `3` are *numeric nodes* whose single output ports stand for the designated number. The nodes labelled `+`, `-`, and `sqrt` are *primop nodes*, whose output ports stand for the result of performing the specified operation on the argument values specified by the input wires.

The node labelled `sink` is a *sink node*, which serves as the primitive source of demand in a computation. Intuitively, a sink node "tugs on" (emits a request to) the wire connected to its single input port in order to initiate a computation. The computation initiated by a sink node successfully terminates when the wire into the sink node enters the returned state. Every computation must have at least one sink node, a distinguished node known as the *top-level sink node*. The top-level sink node loosely corresponds to the `eval` in a traditional Lisp `read-eval-print` loop.

It is worth emphasizing that a snapshot is just a particular moment in the evolution of dynamic process. No mention yet has been made to how a process steps from one snapshot to the next. Nevertheless, it is possible to deduce certain aspects of the history of a process from the wire states in a single snapshot. In Figure 8.1, for example, the wire annotations indicate that demand has propagated from the `sink` node through the `+` node to the `sqrt` node (where demand stops) and to the `-` node (through which demand has propagated to the `7` and `3` nodes). Furthermore, the `3` node has returned to the `-` node in response to the demand.

We shall soon see how the allowable transitions between snapshots are constrained in order to guarantee that a sensible history can be deduced from the wire states of any particular shapshot.

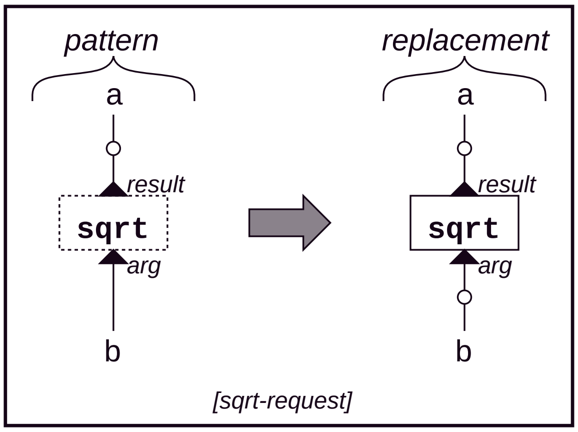


Figure 8.2: A rule that propagates demand through a `sqrt` node.

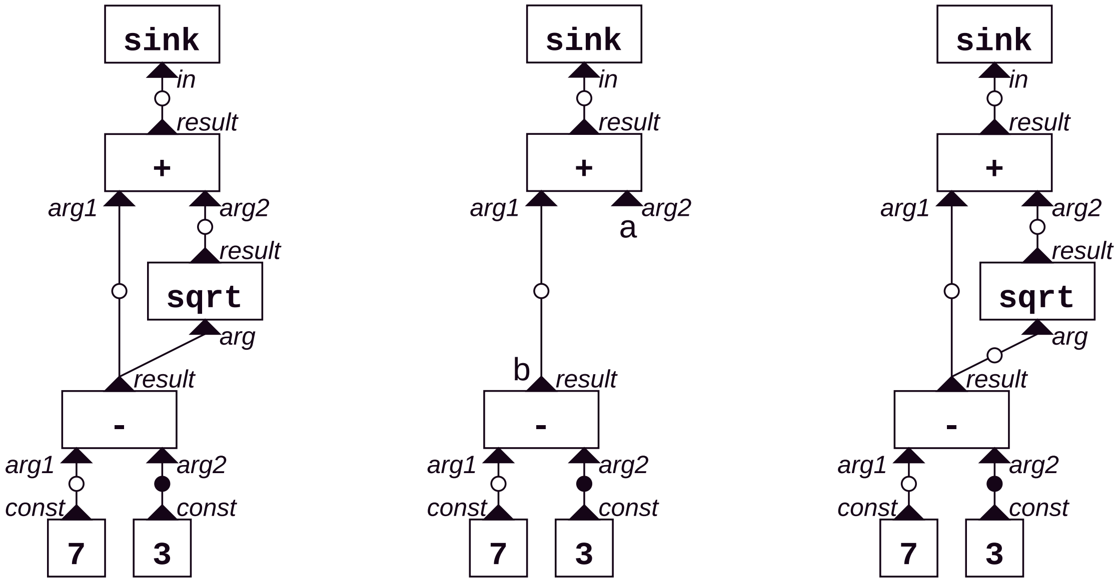
8.1.2 Rewrite Rules

The dynamic behavior of a computation is specified by a set of *rewrite rules* that transform one snapshot into another. It is the rewrite rules that dictate the dynamic behavior of nodes and the flow of demand and values through a sequence of snapshots.

A rewrite rule has two parts, a *pattern* and a *replacement*. The pattern is a partial graph structure in which wires may be attached to *pattern variables* instead of ports. A pattern is said to *match* a snapshot if it can be embedded in the snapshot. The replacement specifies how the snapshot structure matched by the pattern should be replaced in order to construct a new snapshot.

Figure 8.2 shows a simple rewrite rule for propagating demand through a `sqrt` node. The rule says that if a snapshot contains a `sqrt` node having at least one output wire in the requested state and its input wire in the inactive state, then the snapshot can be transformed into a new graph with the same structure as the original except that the input wire is in the requested state. The rule has been labelled with a name ([*sqrt-request*]) so that we can conveniently refer to it later.

A rewrite rule that matches a snapshot can be *applied* to the snapshot to yield a new snapshot. Removing the structure specified by a pattern from a snapshot that it matches leaves the *context* of the match. The new snapshot is constructed from the context by



(a) A snapshot matched by the [*sqrt-request*] rule.

(b) The context determined by the match.

(c) The result of the rule application.

Figure 8.3: Steps in an application of the [*sqrt-request*] rule.

filling the hole left by the deleted pattern with the structure specified by the replacement. The part of the initial snapshot that is not directly matched by the pattern is carried over unchanged into the final snapshot.

For example, the pattern of the [*sqrt-request*] rule matches the snapshot in Figure 8.3(**a**). The context determined by the match appears in Figure 8.3(**b**). Context ports that match a pattern variable in the pattern are labelled to indicate the match. The structure specified by the rule replacement is "glued" onto the context to form the snapshot that is the result of the rule application (Figure 8.3(**c**)). As shown in Figure 8.4, a rule application can be depicted by showing both the original snapshot and the resulting snapshot. (Henceforth, we will omit names on ports by assuming that input and output ports on a node appear in a canonical left-to-right order.)

Each rule pattern contains a distinguished node-matching element called its *locus*. In figure for the [*sqrt-request*] rule, the locus of the rule is the `sqrt` node, which is indicated by a dotted outline. The locus provides a convenient way to specify a particular node in a snapshot at which the match occurs. In a successful match, the node matched by the locus of a rule is said to be *enabled* by the rule. Whenever a snapshot contains a node enabled

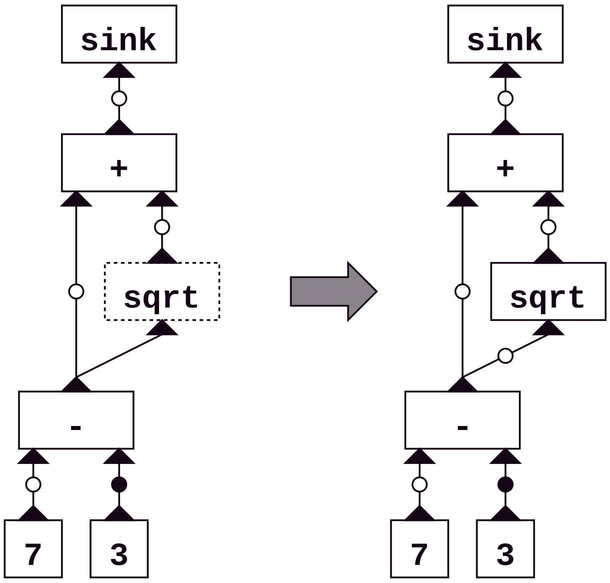


Figure 8.4: An application of the [*sqrt-demand*] rule. Port labels have been dropped here (and in following figures) because they are unambiguously determined by position.

by a rule, the rule can be applied at that node to yield a new snapshot. In Figure 8.4, the dotted outline of the `sqrt` node indicates that it is enabled by the [*sqrt-request*] rule.

Figure 8.5 shows all the EDGAR rules necessary for handling the kinds of nodes we have seen so far. The [*sink-request*] rule shows that a sink acts as a source of demand in the system. Although other nodes may propagate demand, sinks are EDGAR's only source of demand.

The [*constant-return*] rule is the only rule for handling constants (e.g., numbers, booleans, primitive procedures, etc.). Technically, [*constant-return*] is a *rule schema*[3] that stands for an infinite collection of rules; we obtain a particular rule by instantiating the metavariable C with any constant. The [*constant-return*] rule schema dictates that a constant responds to a request simply by returning. The filled-in circle on the output wire of a returned constant indicates that the wire has entered a "returned" state. Unlike certain representations of dataflow, the filled-in circle is just a state indicator and not a value-bearing token; the

[3] I do not emphasize the distinction between rules and rule schemas in the remainder of this discussion, and will sometimes loosely use "rule" where "rule schema" would be more appropriate.

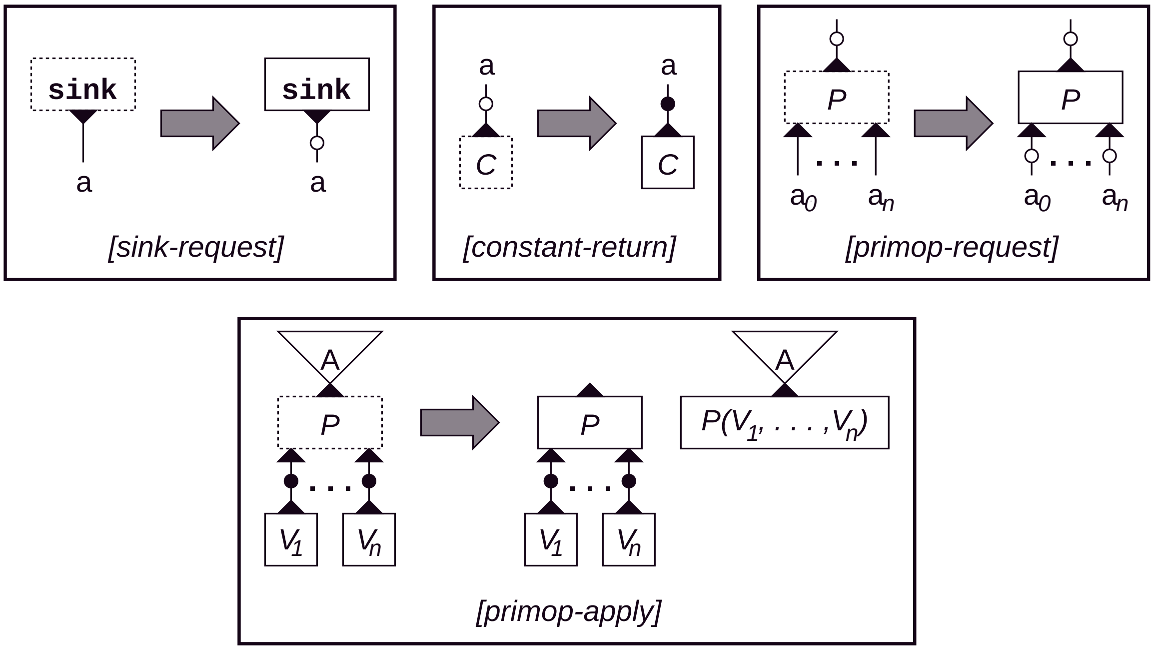


Figure 8.5:

returned value is simply the node at the source of a wire in the returned state.

There are two rules for handling primop nodes. The [*primop-request*] rule specifies that demand for the result of a n-argument primitive operation P propagates *in parallel* to all n arguments of the operation. This rule differs from the operand evalution semantics of most sequential languages, which typically require that the arguments are evaluated in a specified order, or at least some sequential order. The [*primop-request*] rule is one of the means by which concurrency is achieved in EDGAR.

The [*primop-apply*] rule specifies how the application of a primitive operator to values returns a result. An n-argument primop node P is only enabled by such a rule if each of its n input wires is in the returned state (i.e., a primitive application is strict). Each metavariable V_i can be instantiated with any value node that is an acceptable ith argument to P. The node labelled $P(V_1, \ldots, V_n)$ is the result of applying the primitive operator to the argument values. For example, applying a `+` primop node to a `1` node and `2` node would yield a `3` node.

The triangle labelled A at the output port of the primop node is the depiction of a

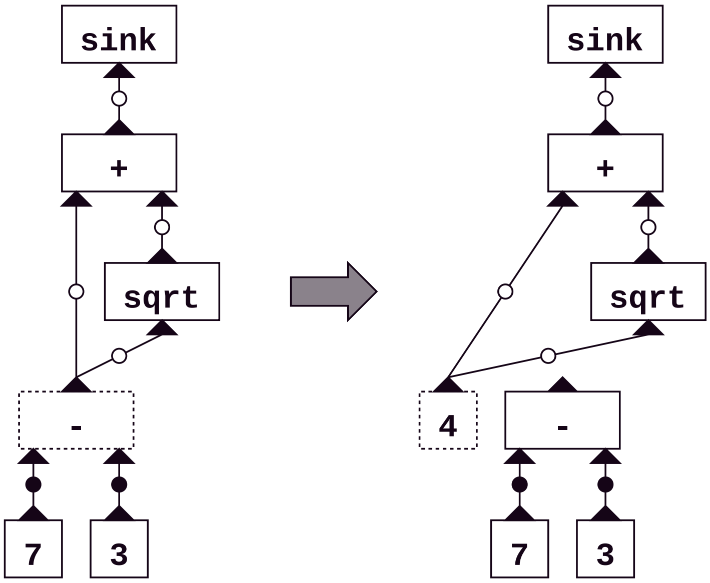


Figure 8.6: Application of a [*primop-apply*] rule for binary subtraction. Note that the `4` node is enabled in the resulting snapshot.

wire set pattern variable. Such a pattern variable matches the *entire* set of wires leaving a port, provided that the set is nonempty and contains at least one requested wire. The [*primop-apply*] rule says that all of the wires originally attached to the P node are "moved" to the output port of the node that is the result of the primitive application. Figure 8.6 shows a sample application of the rule obtained by instantiating the [*primop-apply*] rule with a binary subtraction operator.

8.1.3 Garbage Collection

As indicated by Figure 8.6, rule applications can sometimes result in nodes that are *inaccessible* from the top-level sink node. A node is inaccessible from a sink if there is no directed path of wires from the output port of the node to the input port of the root. In the binary subtraction example, the subgraph rooted at the `-` node is a disconnected component of the snapshot graph, all of whose nodes are inaccessible from the sink.

In order to be able to accurately model the space required by a computation, it is necessary to have some means for removing the inaccessible nodes from a snapshot. This process is called *garbage collection*. We will assume the existence of a garbage collection function, *gc*, that maps snapshots to snapshots by removing any nodes that are not accessible from the sink nodes of a snapshot.[4]

In the case of the [*primop-apply*] rule, the inaccessible nodes in the result are present because the rule specifies that the primop nodes and argument nodes should be copied from the initial snapshot to the final snapshot. Why not change the [*primop-apply*] rule avoid copying these nodes? The problem with this approach is that incorporating garbage collection into the rewrite rules leads to a number of complexities. For example, while it is always safe for the [*primop-apply*] rule to delete the primop node (since it is guaranteed to become inaccessible when all of its output wires are moved), the argument nodes cannot always be deleted since they may be accessible from some other part of the snapshot. Rather than trying to express the garbage collection conditions within the rewrite rules, it is simpler to handle garbage collection via a separate mechanism.

8.1.4 Transitions

Whenever a rule allows snapshot S to be rewritten into S', we say that there is a *transition* between S and $gc(S')$, written $S \Rightarrow gc(S')$. A transition combines a rule application and garbage collection into a single step of the computation. $\Rightarrow$ is a binary relation between snapshots; $\stackrel{*}{\Rightarrow}$ is the reflexive, transitive closure of this relation. When $S_1 \stackrel{*}{\Rightarrow} S_2$, we say that there is a *transition path* from S_1 to S_2.

In general, it may be possible to make several different transitions from a given snapshot. It is often the case that several nodes in a snapshot are enabled by one or more rules. For example, Figure 8.7 shows the two transitions that are possible from the snapshot in Figure 8.1.

Even though more than one rule application may be possible for a given snapshot, transitions are defined so that there is only one rule application per transition. This restriction corresponds to a computational model in which there is a single "processor" that can only apply one rule in each step of the computation. It is possible to imagine alternate approaches

[4]The sink nodes are so-called *roots* of the garbage collection. Eagon nodes are also garbage collection roots.

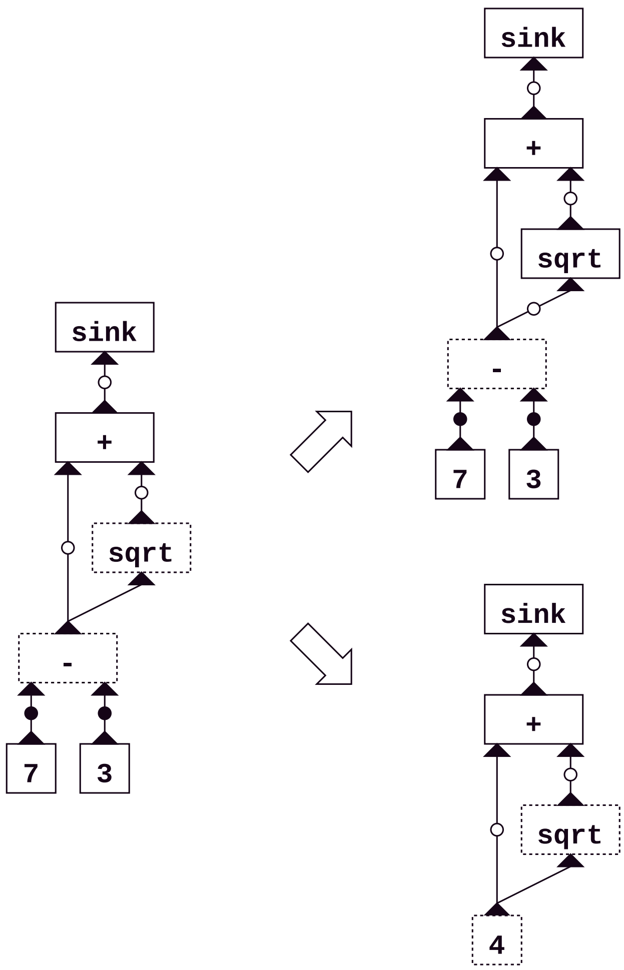


Figure 8.7: The two possible transitions for a snapshot in which two nodes are enabled.

in which more than one rule application would be allowed per transition. For example, a multi-processor model might allow several rule applications to occur in parallel within a single transition. However, allowing multiple applications introduces extra complexities (e.g. undesirable interactions between rules whose patterns might overlap when matching a graph). Since allowing only a single application per transition is sufficient for our purposes, we henceforth ignore the possibility of multiple rule applications per transition.

8.1.5 Computation

Here we formalize the notion of an EDGAR computation. First, a few definitions:

- An *initial snapshot* is a snapshot rooted at a sink node in which all wires are inactive.
- A *final snapshot* is a snapshot in which no rules are enabled.
- A *trace* is any sequence of snapshots such that there is a transition from each element of the sequence to the next.
- The *space* required by a snapshot is the sum of the number of nodes and the number of wires in the snapshot.

A *terminating computation* is a trace from an initial snapshot to a final snapshot. The *time* required by a terminating computation is the number of transitions made in the trace (i.e., one less than the length of the trace). The *space* required by a terminating computation is the maximum of the space of the snapshots in the computation.

For example, Figure 8.8 depicts a sample terminating computation as a "movie" of numbered snapshot "frames". The time of the sample computation is 13 transitions, and the space is 12 units (the first 6 frames require 6 nodes + 6 wires).

A *non-terminating computation* is an infinite trace beginning with an initial snapshot. The *time* required by a non-terminating computation is undefined. The *space* required by a non-terminating computation is the maximum of the space of the snapshots in the computation *if* this maximum is defined; otherwise the space is undefined.

A computation has an *outcome* that describes its fate. The outcome of a non-terminating computation is *bottom*. The outcome of a terminating computation is determined by its

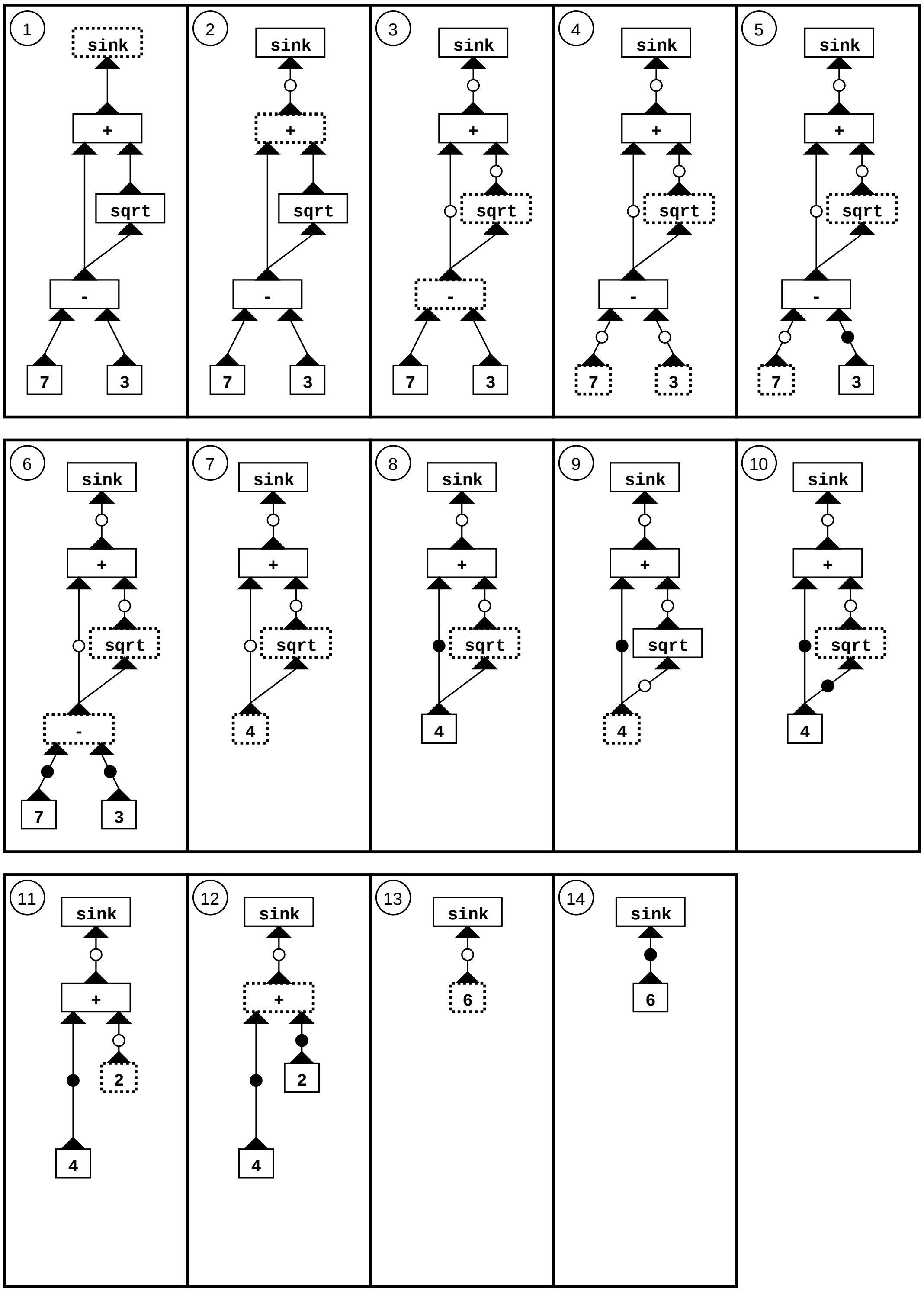


Figure 8.8: Depiction of a computation as a sequence of snapshots.

final snapshot. If the input wire to the root sink node is returned, then the outcome is a *result* whose value is the graph structure at the source of the wire. For example, the outcome of the computation depicted in Figure 8.8 is a result whose value is the constant node `6`. If the input wire to the root sink is not returned, then the outcome is *deadlock*. Deadlock indicates a computation that is "stuck" in a snapshot in which no rules are applicable. Figure 8.9 shows two final snapshots of a deadlocked computation. In **(a)**, the snapshot corresponds to a type error (`sqrt` cannot be applied to a boolean); in **(b)**, the snapshot is caught in an unresolvable dependency.

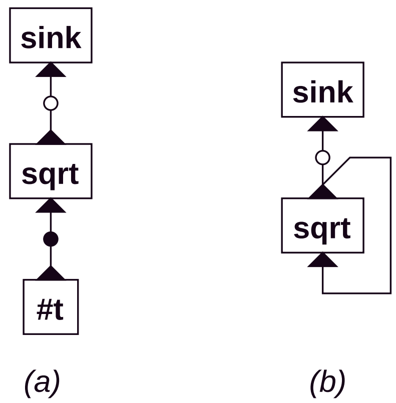
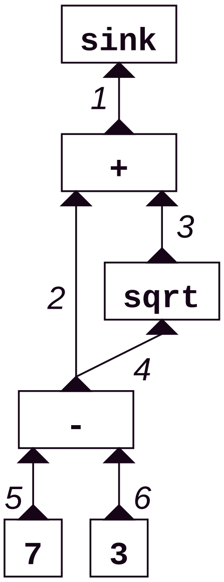


Figure 8.9: Two examples of deadlocked final snapshots.

8.1.6 Behavior

The *behavior* of an initial snapshot is the set of all computations that begin with that snapshot. A behavior often contains numerous computations due to the fact that several transitions may be possible from a given snapshot. The *time* and *space* required by of a behavior can be found by maximizing over the spaces and times required by the component computations (when these quantities are defined). The *outcomes* of a behavior is the set of outcomes for the component computations. Due to the fact that EDGAR supports side effects (data mutation and I/O), the nondeterminism of transitions can lead to multiple outcomes.

Consider the initial snapshot $S_{initial}$ used in Figure 8.8:



The wires in $S_{initial}$ have been labelled so that we can refer to them by number. The behavior of $S_{initial}$ is depicted in Figure 8.10 as a directed graph in which each vertex stands for a snapshot and each directed edge stands for a transition. Each vertex is a triple of numeric sets indicating the states of the wires labelled by the numbers. The triple is interpreted as

$$\langle inactive\ wires, requested\ wires, returned\ wires\rangle$$

A number can be a member of at most one of the three sets; if it is a member of none, then it is not present in the snapshot. For the simple case of $S_{initial}$, a triple uniquely identifies a snapshot in one of $S_{initial}$'s computations. Every path through the graph from $S_{initial}$ to S_{final} represents one of the computations in the behavior of $S_{initial}$. The shaded crosses are isomorphic subgraphs that only differ in whether or not wire 4 is inactive (left cross) or requested (right cross).

8.1.7 Global State

The above definition of behavior assumes that the set of computations emanating from an initial snapshot depends only on the snapshot. Indeed, in the absence of I/O operations or other operations on global state, each initial snapshot identifies a unique behavior. But

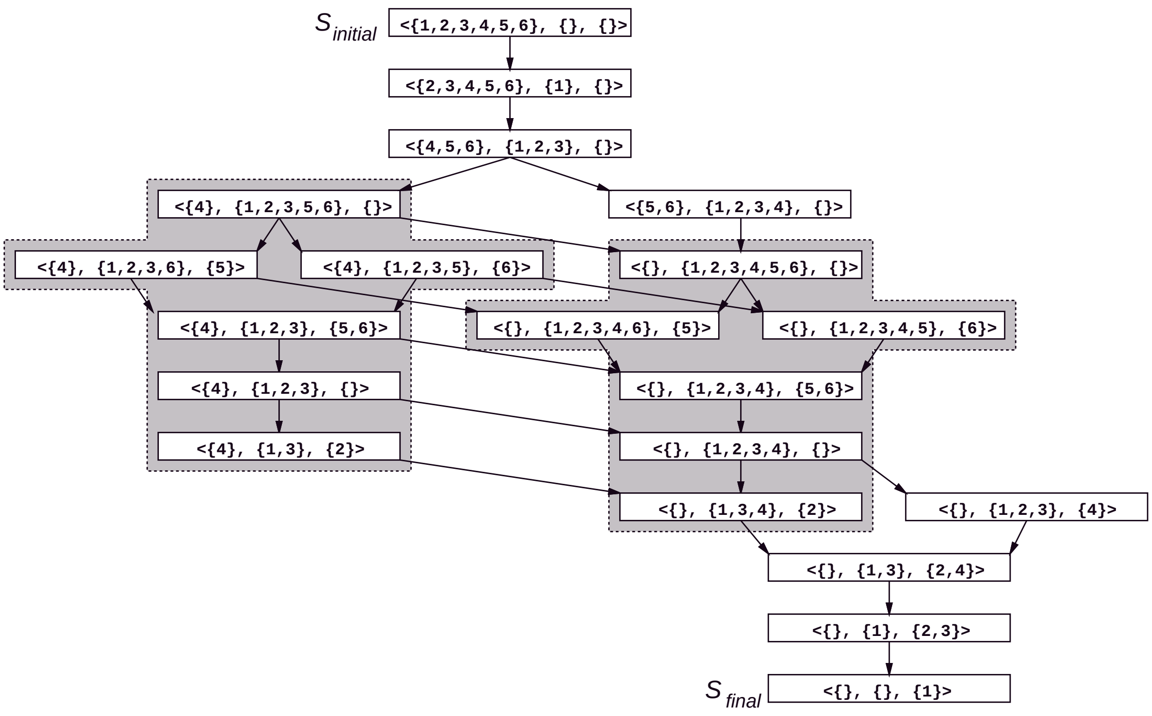


Figure 8.10: Depiction of the behavior of a simple initial snapshot.

in the presence of such operations, the behavior of an initial snapshot may depend on the global state, and individual computations may modify the global state.

There are many approaches to handling global state in the EDGAR framework. Here are some possibilities:

1. Assume that all global state is explicitly encoded in the initial snapshot and can be extracted from the final snapshot.

2. Change the definition of computation so that it is a triple of a trace, an initial state, and a final state such that the trace produces the final state from the initial one. Additionally extend the notion of outcome to include the final state. Behavior would still be defined as a set of computations.

3. Keep the definition of a computation the same, but change the definition of behavior to be a function from initial states to sets of computation/final-state pairs.

I will not pick a particular method, but will simply assume that global state is handled appropriately in situations where it is an issue.

8.2 The Details of EDGAR

Here we detail only those EDGAR rules that are essential for expressing the concurrency, synchronization, and non-strictness features of OPERA. The complete collection of EDGAR rules is described in [Tur94].

8.2.1 Procedures

The handling of procedures and procedure calls is shown in Figure 8.11. A `pcall` node is a general procedure application node. The [*pcall-request*] rule propagates demand for a `pcall` node to all of its input wires. Together, the [*primop-request*] rule (from the previous section) and the [*pcall-request*] rule implement a concurrent strategy for argument evaluation.

Nodes labelled `proc` represent procedures. Each `proc` node contains a *template*, a partial graph structure that specifies the body of the procedure. The template has an output port that represents the result of the body, and a number of input ports that represent the

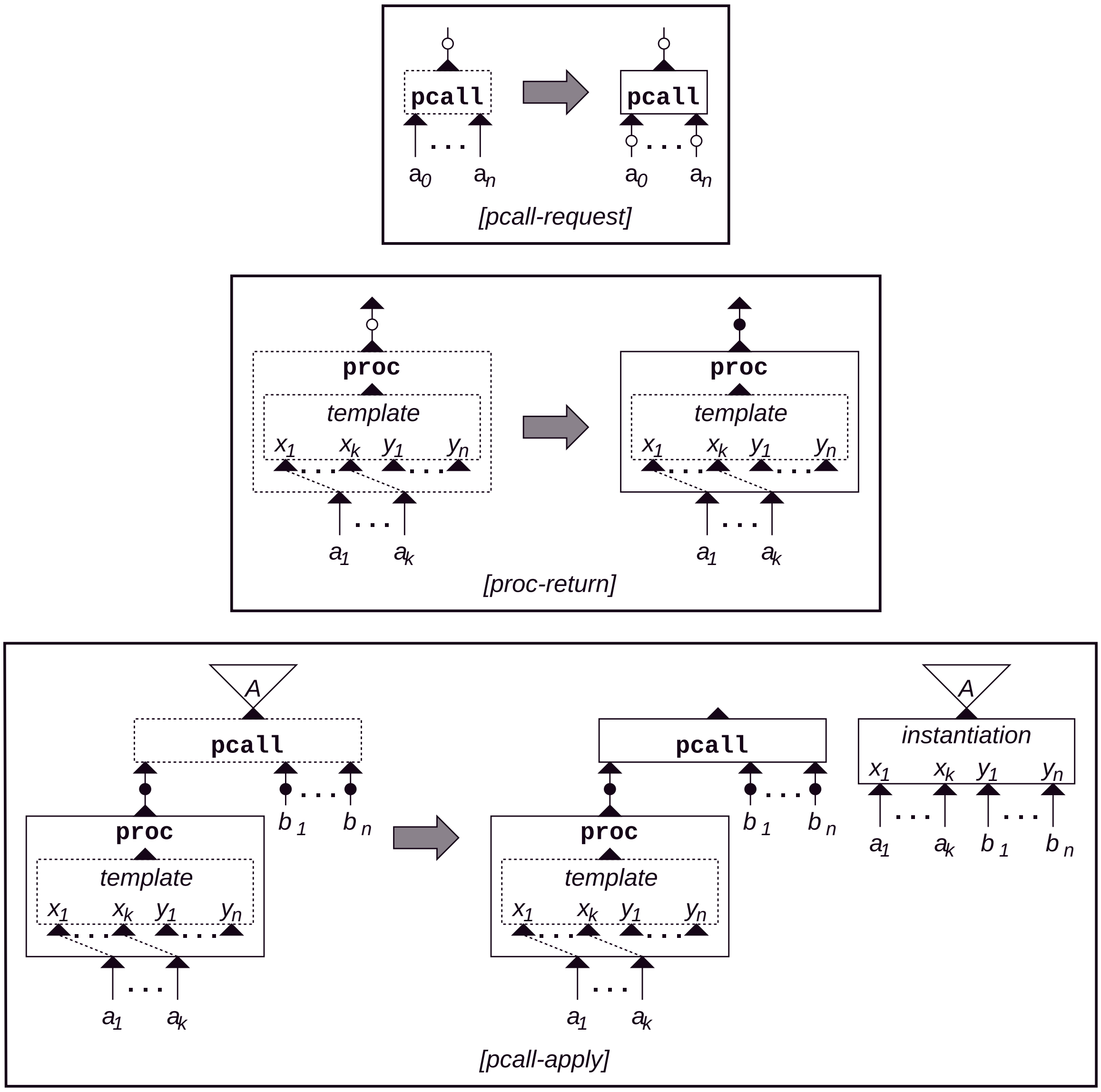


Figure 8.11: EDGAR rules for handling procedures.

values on which the body depends. The k input ports x_1, ..., x_k stand for the values of free variables used in the body, while the n input ports y_1, ..., y_n stand for the explicit arguments of the procedure. The `proc` node itself has k input ports that are attached to the values of the free variables (a_1, ..., a_k). These inputs correspond to the environment of a closure in the traditional environment model of procedures [ASS85]. The dotted lines connected the first k input ports on the template to the input ports of the `proc` node are intended to suggest that the template inputs will be connected to the `proc` inputs upon application of the procedure.

The [*proc-return*] rule simply says that a `proc` node is a value. The [*pcall-apply*] rule specifies the details of procedure application. The rule is not applicable until all the input wires are returned, which indicates that procedure calls are strict. The rule creates a copy of the template, called the *instantiation*, and connects wires to its ports as follows:

- The first k input wires of the instantiation are connected to the sources of the k input wires of the `proc` node. This supplies the values of the free variables as implicit arguments to the instantiation.

- The last n input wires of the instantiation are connected to the sources of the n operand wires of the `pcall` node. This supplies the values of the explicit arguments to the instantiation.

- All of the output wires of the `pcall` node are rerouted to the output port of the instantiation. This means that the value of the instantiation will be "returned" as the value of the `pcall` node.

8.2.2 Synchrons

I present two versions of the synchron rules. The simpler version, which ignores the complexities of the `precede!` operator, is shown in in Figure 8.12. The [*synchron-return*] rule treats `synchron` nodes as self-evaluating values. The [*simultaneous*] rule is a unification rule that ensures that all references to two unified synchrons point to the same object.

The [*rendezvous*] rule is the cornerstone of the OPERA semantics. In fact, the whole EDGAR system was design to express this rule in a reasonable way! A synchron output wire

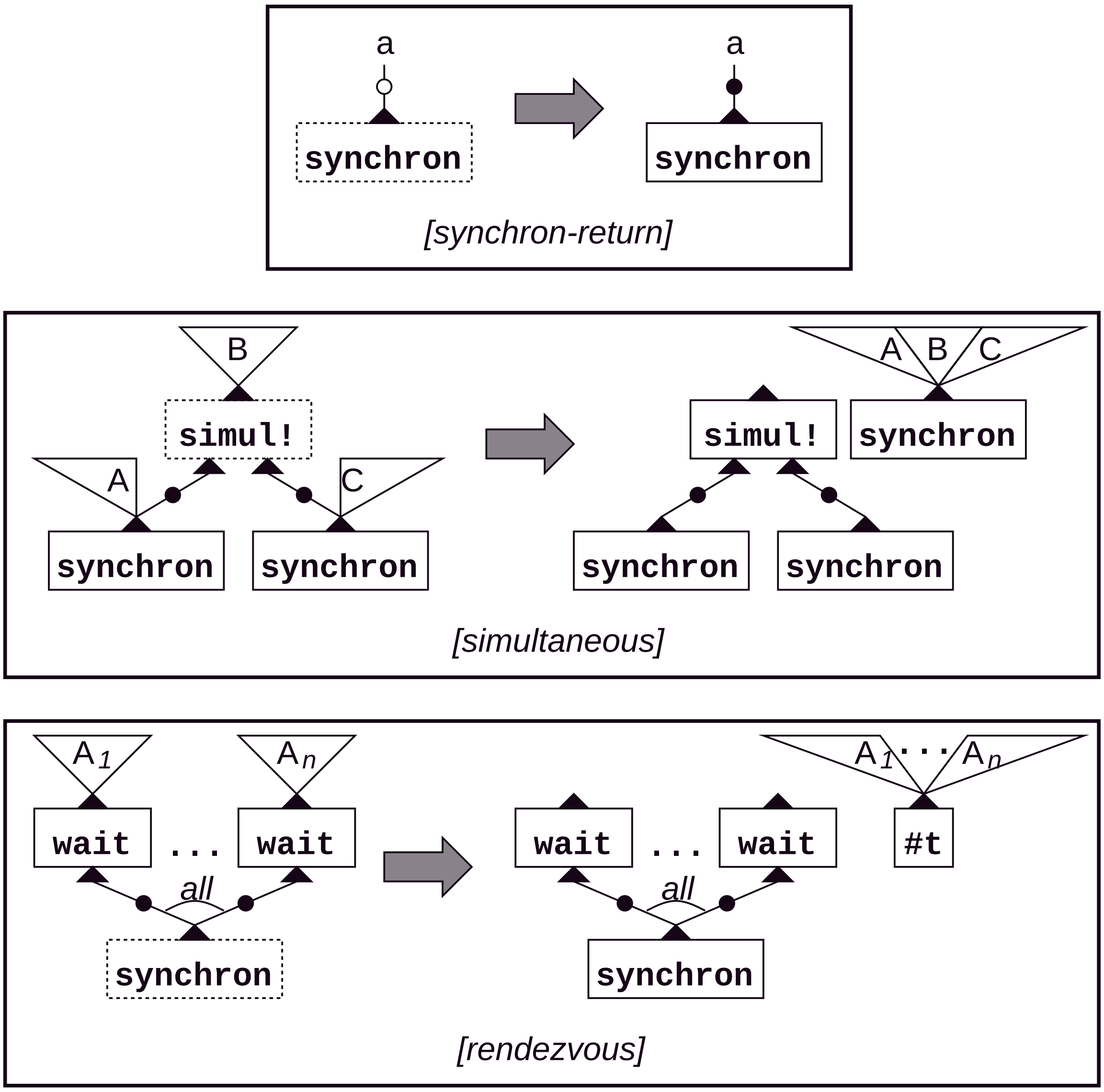


Figure 8.12: Simple versions of the EDGAR synchron rules.

that is in the returned state and attached to a demanded `wait` node is a *waiting pointer*. Any other output wire of a synchron is a *non-waiting pointer*. The rendezvous rule is only enabled when all of the output wires of a synchron are waiting pointers. The result of the rule is that a constant true node is returned to all output wires of all the wait nodes involved in the rendezvous. Note that the synchron is inaccessible after the [*rendezvous*] rule, and can be garbage collected. The rule essentially embodies a proof that the rendezvous is safe since every process that *could ever* wait on the synchron *is* waiting on the synchron.

The `precede!` operator complicates the handling of synchrons. The constraint that synchron A precedes synchron B will be represented by augmenting A with an input port that is connected to the output port of B via an inactive wire. Since the wire is a non-waiting pointer, B is blocked from rendezvousing as long at A is still accessible.

The modified rules necessary for handling synchrons in the presence of `precede!` are shown in Figure 8.13. Each synchron is assumed to have a set of input wires. The [*precede*] rule adds a new input wire to one synchron only if the other is not already a follower; the result returned by the `precede!` node is a constant true node. The [*simultaneous*] rule is modified to take the union of the followers of the two synchrons being combined. The [*rendezvous*] rule is trivially updated to indicate that nothing happens to the followers.

One last wrinkle concerning synchrons in the presence of `precede!` is that the *gc* garbage collection function must be modified to transform certain configurations of connected synchrons in order for OPERA to maintain the right storage behavior. The transformation is expressed as the [*compaction*] pseudo-rule in Figure 8.14. (Although expressed as a rewrite rule, the transformation is part of the *gc* garbage collection function, not the rewrite system). The pseudo-rule says that if a synchron with a single follower is only pointed at by other synchrons, it can safely be removed. This reduces the amount of space required by chains of unreferenced synchrons, which can occur in the *up* processing of filtered lists (see Section 5.5.4). Extending the rule to synchrons with more than one follower is not a good idea, because the transformation would actually *increase* the space by creating new wires.

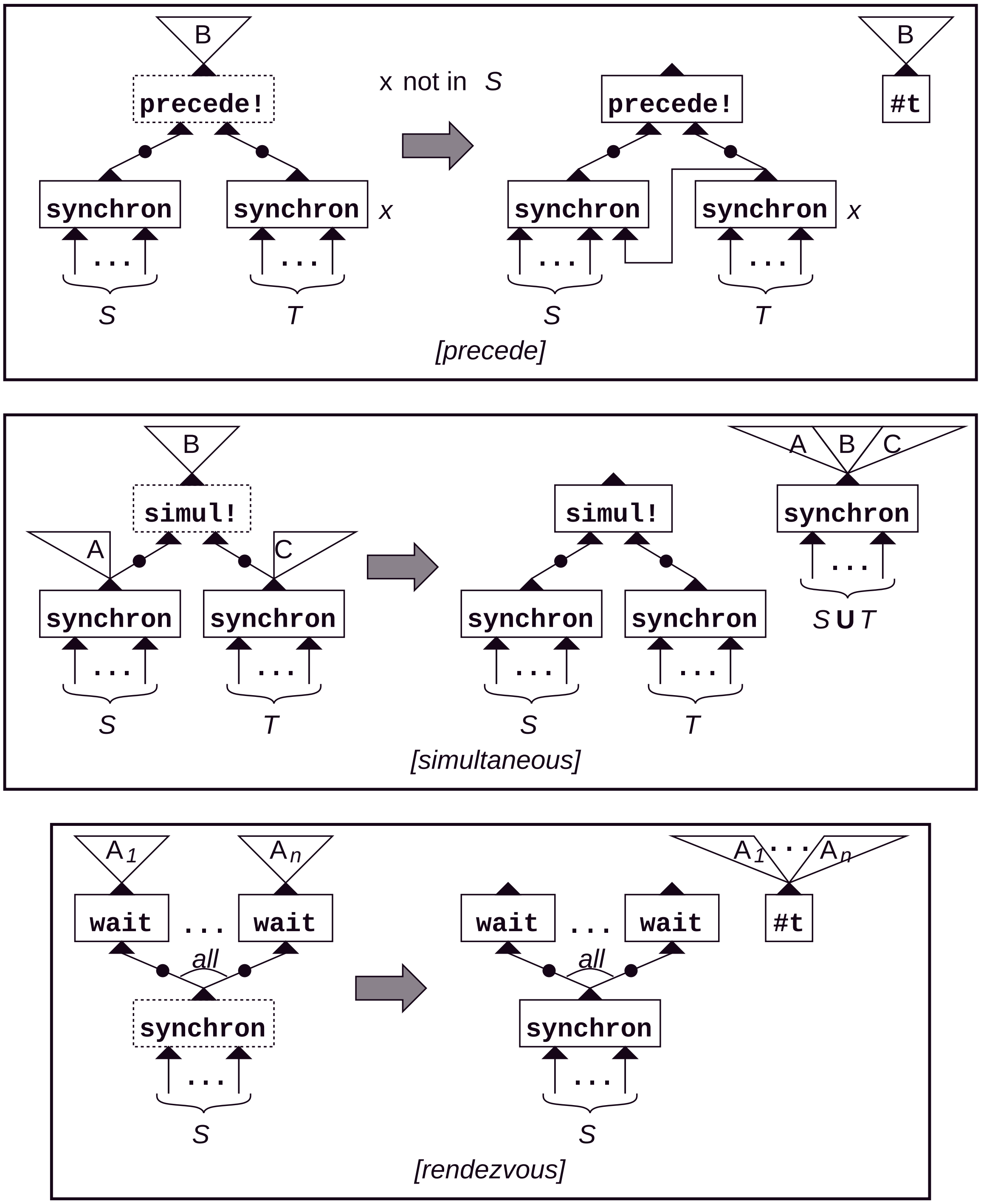


Figure 8.13: Extended versions of the EDGAR synchron rules.

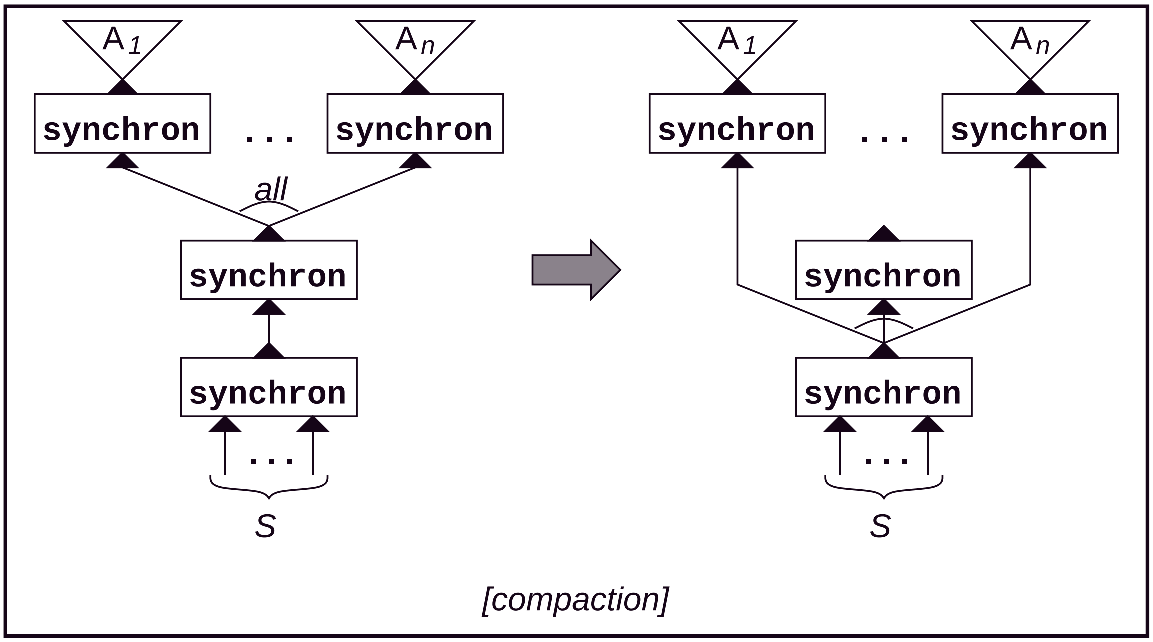


Figure 8.14: Pseudo-rule for synchron garbage collection that must be implemented by the *gc* function.

8.2.3 Excludons

In Opera, excludons and the `exclusive` construct are responsible for implementing mutual exclusion. The rules for manipulation excludons are shown in Figure 8.15. The [*excludon-return*] rule indicates that excludons are values. The side conditions on this rule implement the locking behavior of excludons; if an excludon has a returned output wire to one `exclusive`, it cannot return to another. The [*exclusive-request-lock*] rule requests an excludon, but this process will be blocked if the excludon has a returned wire to another `exclusive`. Once an `exclusive` has obtained a lock, it can evaluate the body ([*exclusive-with-lock*]). When the body returns ([*exclusive-release-lock*]), the lock is effectively released because garbage collection will remove the returned wire to the inaccessible `exclusive`.

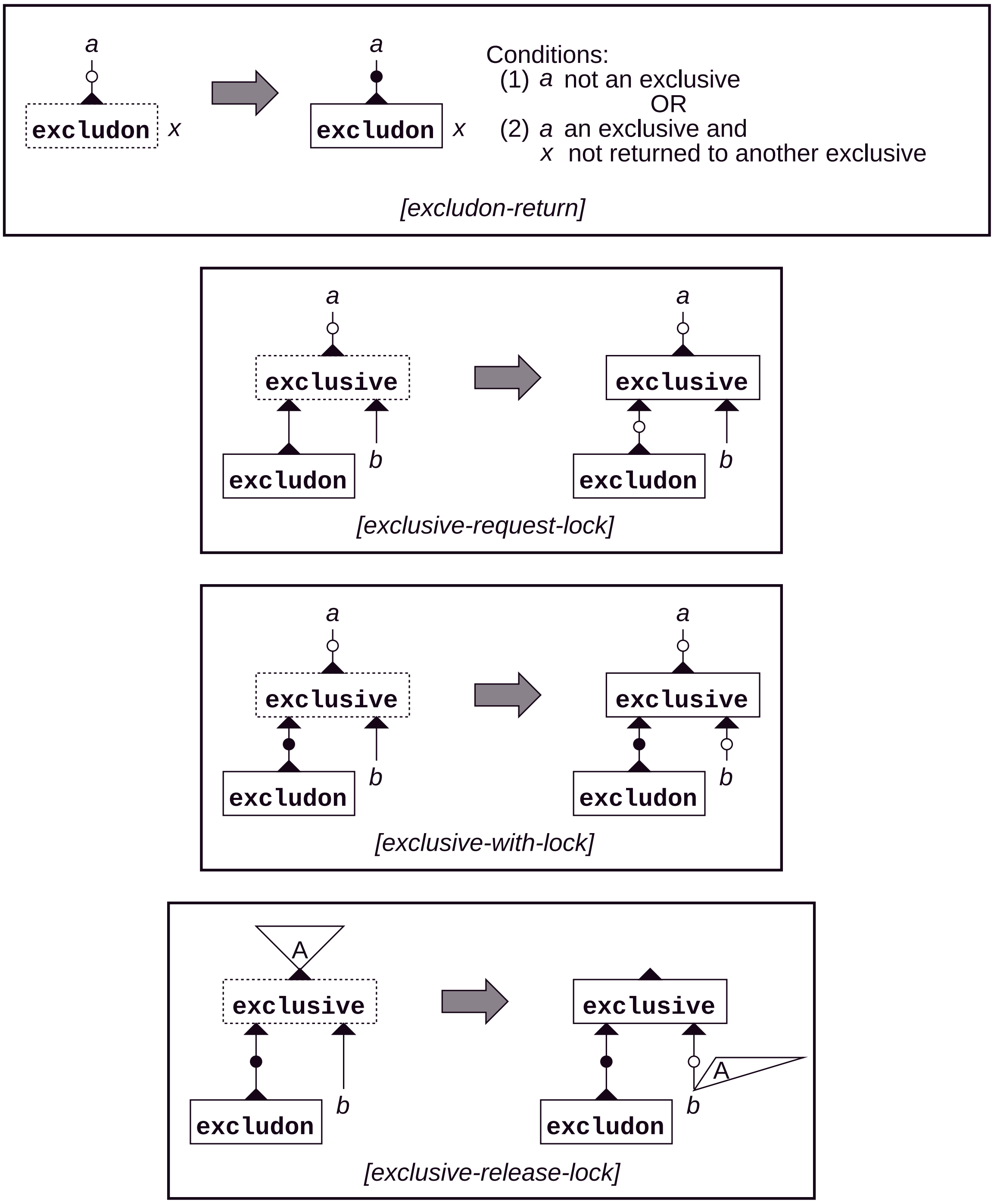


Figure 8.15: Rules for handling `excludons`.

8.2.4 Lazons and Eagons

OPERA's non-strictness is expressed in terms of lazons and eagons. The EDGAR rewrite rules for handling these to objects are shown in Figure 8.16. The rules indicate there is only a single difference between lazons and eagons: a lazon returns to a request without propagating the demand, while an eagon returns to a request *and* propagates demand. The dotted input wire to the `eagon` node in the [*eagon-touch*] rule indicates that the wire may be in any state.

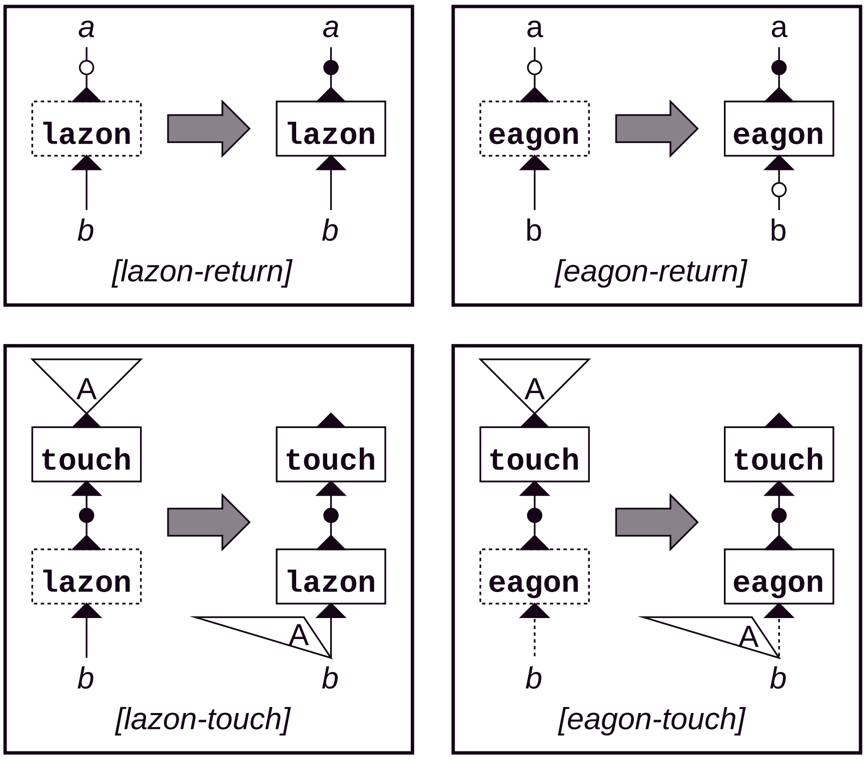


Figure 8.16: EDGAR rules for handling lazons and eagons.

8.3 Compiling OPERA into EDGAR

The semantics of OPERA can be formalized by describing how to translate an OPERA program into an EDGAR initial snapshot. The behavior of this snapshot is the meaning of the OPERA expression. The results of this snapshot are the possible values of the expression.

To simplify the presentation, we will introduce an intermediate language, OK, which is a slight variant of the OPERA kernel. We will then present the OPERA to EDGAR translation process in two stages: the translation from OPERA to OK, and the translation from OK to EDGAR.

8.3.1 OK

OK is a language that is a minor tweak of the OPERA kernel. The syntax of OK is summarized in Figure 8.17. (For comparison, see Figure 7.1 on page 296 for a summary of the OPERA kernel.) OK supports almost all the kernel syntax of OPERA. The differences are as follows:

- Unlike an OPERA program, an OK program includes no definitions. It consists of a single OK expression.
- Whereas OPERA's `quote` form can express arbitrary symbolic expressions as literals, OK's `quote` form can only express symbols.
- OK does not support the assignment form `(set!` I_{name} E_{val}`)`.
- OK includes the new form `(primop` O_{prim} E_{arg}`*)` to express the application of primitive operators. O_{prim} must be the name of one of OPERA's primitive procedures. `(primop + 1 2)` can be viewed as an "in-lined" version of `(+ 1 2)`.
- The forms `(seq1` E`*)` and `(seqn` E`*)`, which are syntactic sugar in OPERA, are considered to be kernel forms in OK.
- Except for the `seq1` and `seqn` forms, OK does not support any of the syntactic sugar of OPERA.

8.3.2 Translating OPERA to OK

The translation from OPERA to OK is performed by two functions:

1. $\mathcal{T}_{\text{prog}}$ maps an OPERA program to an OK program.

Kernel Grammar:

P	$\in$	Program	
E	$\in$	Expression	
I	$\in$	Identifier	
L	$\in$	Literal	
S	$\in$	Symbolic Expression	
O	$\in$	Operator Name	
P	::=	(program E_{body})	[program]
E	::=	L	[literal expression]
	|	I	[variable reference]
	|	(lambda (I_{formal}*) E_{body})	[abstraction]
	|	(pcall E_{proc} E_{arg}*)	[parallel application]
	|	(pletrec ((I_{name} E_{def})*) E_{body})	[parallel recursive bindings]
	|	(if E_{test} E_{then} E_{else})	[conditional]
	|	(quote I)	[quoted expressions]
	|	(lazon E_{body})	[suspension]
	|	(eagon E_{body})	[future]
	|	(exclusive E_{excl} E_{body})	[mutual exclusion]
	|	(nex ((I_{name} E_{def})*) E_{body})	[graphical bindings]
	|	(nexrec ((I_{name} E_{def})*) E_{body})	[graphical recursive bindings]
	|	(primop O_{prim} E_{arg}*)	[primitive application]
	|	(seq1 E_{sequent}*)	[sequence/first]
	|	(seqn E_{sequent}*)	[sequence/last]
I	::=	*usual Scheme identifiers*	
L	::=	*usual Scheme literals*	

Figure 8.17: OK summary

2. $\mathcal{T}_{\text{exp}}$ maps an OPERA expression to an OK expression.

$\mathcal{T}_{\text{prog}}$ simply transforms an OPERA program into an OK program whose body is a `pletrec` expression:

$$\mathcal{T}_{\text{prog}}[\![\texttt{(program } E_{\text{body}} \texttt{ (define } I\ E\texttt{) ...)}]\!]$$
$$= \texttt{(program } \mathcal{T}_{\text{exp}}[\![\texttt{(pletrec ((}I_1\ E_1\texttt{) ...) } E_{\text{body}}\texttt{)}]\!]\texttt{)}$$

The $\mathcal{T}_{\text{exp}}$ function can be expressed as the composition of a number of source-to-source transformations I present each component transformation below in the order in which they are applied. I will treat each transformation as a separate phase, though in practice the phases can be interwoven into a transformation program that makes fewer passes over the OPERA expression (yet another potential application of slivers!):

Desugaring

The desugaring phase, $\mathcal{D}_{\text{exp}}$, transforms an OPERA expression into another OPERA expression by expanding all of the syntactic sugar forms. The syntactic sugar inherited from Scheme (except for `begin`) is expanded according to the rules given in [CR+91]. The OPERA-specific syntactic sugar is trivially expanded as follows:

$$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E \ldots\texttt{)}]\!] = \texttt{(seqn } \mathcal{D}_{\text{exp}}[\![E]\!] \ldots\texttt{)}$$

$$\mathcal{D}_{\text{exp}}[\![\texttt{(}E_{\text{proc}}\ E_{\text{arg}} \ldots\texttt{)}]\!] = \texttt{(pcall } \mathcal{D}_{\text{exp}}[\![E_{\text{proc}}]\!]\ \mathcal{D}_{\text{exp}}[\![E_{\text{arg}}]\!] \ldots\texttt{)}$$

$$\mathcal{D}_{\text{exp}}[\![\texttt{(let ((}I\ E\texttt{) ...) } E_{\text{body}}\texttt{)}]\!]$$
$$= \texttt{(pcall (lambda (}I \ldots\texttt{) } \mathcal{D}_{\text{exp}}[\![E_{\text{body}}]\!]\texttt{) } \mathcal{D}_{\text{exp}}[\![E]\!] \ldots\texttt{)}$$
; `plet` is handled similarly

$$\mathcal{D}_{\text{exp}}[\![\texttt{(letrec ((}I\ E\texttt{) ...) } E_{\text{body}}\texttt{)}]\!]$$
$$= \texttt{(pletrec ((}I\ \mathcal{D}_{\text{exp}}[\![E]\!]\texttt{) ...) } \mathcal{D}_{\text{exp}}[\![E_{\text{body}}]\!]\texttt{)}$$

$$\mathcal{D}_{\text{exp}}[\![\texttt{(seq } E \ldots\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![\texttt{(seq1 \#t } E \ldots\texttt{)}]\!]$$

Though simple expansions exist for `seq1` and `seqn`, these forms are left unexpanded by the desugaring phase.

Quote removal

The quote removal phase, $\mathcal{Q}_{\text{exp}}$, transforms one OPERA expresion to another by lifting compound quoted expressions to top level and expanding them. A quoted expression is compound if the text of the quotation is a list or vector.

The lifting stage names all compound quoted expressions at top level and replaces their former positions with a variable reference. For example, lifting would transform:

```
(lambda () (list '(a b) 'd '(e (f g))))
```

into[5]

```
(let ((%1 '(a b))
      (%2 '(e (f g))))
  (lambda () (list %1 'd %2)))
```

The expansion stage, $\mathcal{X}_{\text{sexp}}$, uses the following transformations to expand quoted expressions:

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'}L]\!] = L$$

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'(}S\ \ldots\texttt{)}]\!] = \texttt{(list } \mathcal{X}_{\text{sexp}}[\![\texttt{'}S]\!]\ \ldots\texttt{)}$$

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'(}S_{\text{car}}\ \texttt{.}\ S_{\text{cdr}}\texttt{)}]\!] = \texttt{(cons } \mathcal{X}_{\text{sexp}}[\![\texttt{'}S_{\text{car}}]\!]\ \mathcal{X}_{\text{sexp}}[\![\texttt{'}S_{\text{cdr}}]\!]\texttt{)}$$

$$\mathcal{X}_{\text{sexp}}[\![\texttt{'\#(}S_{\text{car}}\ \ldots\texttt{)}]\!] = \texttt{(vector } \mathcal{X}_{\text{sexp}}[\![\texttt{'}S_{\text{car}}]\!]\ \ldots\texttt{)}$$

$\mathcal{X}_{\text{sexp}}$ resembles the expander associated with Lisp's backquote notation, except that it works on quote notation instead. It leaves behind a tree of data constructors that will later create the structure denoted by the quoted expression. The leaves of the tree are literals and quoted symbols (quoted symbols are left untouched by $\mathcal{X}_{\text{sexp}}$).

Quote removal is necessary to accurately model Lisp's treatment of quoted expressions within EDGAR. In Lisp, the data structure of a quoted expression is created by the Lisp reader; every evaluation of the quoted expression returns this same data structure. In particular, the data structure can be mutated. Consider the following Scheme procedure:

[5] I use a `let` here to enhance readability, but since the $\mathcal{Q}_{\text{exp}}$ transformation is performed after the desugaring phase, the actual transformation needs to use the desugared form of `let`. In the interests of readability, I will similarly make use of other unavailable constructs in later examples; but in all cases, the transformation can be made precise without the offending constructs.

```
(let ((foo (lambda () '(1 2 3))))
  (begin (set-car! (foo) 17)
         (foo))))
```

The value of the above expression should be `(17 2 3)` because the `set-car!` mutates the value of the quoted list in the body of the `foo` procedure. Quote removal expresses this behavior by constructing the data structure for a quoted expression at top level (simulating the Lisp reader) and using naming to appropriately share the resulting value.

Assignment conversion

Since EDGAR does not support any kind of named entities, all names must be removed as part of the compilation process. Assignment conversion is one aspect of name removal. The assignment conversion phase, $\mathcal{A}_{\text{exp}}$, transforms OPERA expressions to OK expressions by removing all assignments — i.e., expressions of the form `(set!` I E`)`. Assignment conversion is described in [K+86]. The basic idea behind assignment conversion is to transform all assignments into mutation operations on cells (mutable one-slot data structures).

As an example, consider the OPERA expression:

```
(lambda (count)
  (lambda (inc)
    (seqn (set! count (+ count inc))
          count)))
```

The following OK expression results from applying the assignment conversion function, $\mathcal{A}_{\text{exp}}$, to the above OPERA expression:

```
(lambda (count)
  (let ((%count (cell count)))
    (lambda (inc)
      (seqn (cell-set! %count (+ (cell-ref %count) inc))
            (cell-ref %count)))))
```

The name `count` introduced by the `lambda` is shadowed by the `count` introduced by the `let`. The latter is bound to a cell containing the value of the former. The assignment to the former `count` is replaced by a `cell-set!` operation on the latter. Each variable reference to the former count is replaced by a `cell-ref` operation on the latter. Note that the `inc` variable, which is not assigned to, is left untouched.

In general, assignment conversion processes each name introduced by a `lambda` or `pletrec`. If the name is not assigned to in its lexical scope, it is left alone (like `inc`).

However, if the name is assigned in its lexical scope, then (1) a new variable is introduced and bound to a cell holding the value of the original; (2) every reference to the name in its lexical scope is replaced by an appropriate call to `cell-ref`; and (3) every assignment to the name in its lexical scope is replaced by an appropriate call to `cell-set!`.

Note that assignment conversion does *not* process names introduced by a `nex` or a `nexrec`. In fact, attempting to assign to a variable introduced by one of these constructs is an error.

Eta expansion

The eta expansion phase, $\mathcal{E}_{\mathrm{exp}}$, maps an OK expression to an OK expression by eta expanding [Bar84] every reference to a primitive procedure that does not occur in the operator position of a `pcall` application. For example:

$$\mathcal{E}_{\mathrm{exp}}[\![\texttt{(pcall f + (pcall + 1 2))}]\!]$$
$$= \texttt{(pcall f (lambda (a b) (pcall + a b)) (pcall + 1 2))}$$

Here, the first instance of `+` does not occur in the operator position of a pcall, so it is expanded in a `lambda` of two arguments (we assume that `+` takes exactly two arguments). Note that the expansion moves `+` into the operator position of a `pcall`. At the end of the eta expansion phase, every reference to a primitive procedure is in the operator position.

The name of a primitive procedure is only eta expanded if is not shadowed by a lexical variable. For instance, eta expansion acts as the identity on `(lambda (+) (f + 1))` because the name `+` introduced by the `lambda` is not a reference to the primitive procedure `+`.

In-lining of primitives

The in-lining phase, $\mathcal{I}_{\mathrm{exp}}$, maps OK expressions to OK expressions by converting every application of a primitive operator into a `primop` expression:

$$\mathcal{I}_{\mathrm{exp}}[\![\texttt{(pcall } O_{\mathrm{prim}}\ E_{\mathrm{arg}}\ \ldots\texttt{)}]\!] = (\mathrm{primop}\ O_{\mathrm{prim}}\ E_{\mathrm{arg}}\ \ldots)$$

This phase merely tags primitive operator applications as being special.

Touch introduction

The touch introduction phase, $\mathcal{H}_{\text{exp}}$, wraps an explicit `touch` around every expression that occurs in a touching context. The non-trivial clauses for the definition of $\mathcal{H}_{\text{exp}}$ are shown below:

$$\begin{array}{l}
\mathcal{H}_{\text{exp}}[\![\texttt{(if } E_{\text{test}}\ E_{\text{then}}\ E_{\text{else}}\texttt{)}]\!] \\
\quad = \texttt{(if (touch } \mathcal{H}_{\text{exp}}[\![E_{\text{then}}]\!]\texttt{) } \mathcal{H}_{\text{exp}}[\![E_{\text{then}}]\!]\ \mathcal{H}_{\text{exp}}[\![E_{\text{else}}]\!]\texttt{)} \\[1em]
\mathcal{H}_{\text{exp}}[\![\texttt{(pcall } E_{\text{rator}}\ E_{\text{rand}}\ \ldots\texttt{)}]\!] \\
\quad = \texttt{(pcall (touch } \mathcal{H}_{\text{exp}}[\![E_{\text{rator}}]\!]\texttt{) } \mathcal{H}_{\text{exp}}[\![E_{\text{rand}}]\!]\ \ldots\texttt{)} \\[1em]
\mathcal{H}_{\text{exp}}[\![\texttt{(exclusive } E_{\text{excl}}\ E_{\text{body}}\texttt{)}]\!] \\
\quad = \texttt{(exclusive (touch } \mathcal{H}_{\text{exp}}[\![E_{\text{excl}}]\!]\texttt{) } \mathcal{H}_{\text{exp}}[\![E_{\text{body}}]\!]\texttt{)} \\[1em]
\text{; This clause only applies if } O_{\text{prim}} \text{ is neither a constructor nor } \texttt{touch}. \\
\mathcal{H}_{\text{exp}}[\![\texttt{(primop } O_{\text{prim}}\ E_{\text{arg}}\ \ldots\texttt{)}]\!] \\
\quad = \texttt{(primop } O_{\text{prim}}\ \texttt{(touch } \mathcal{H}_{\text{exp}}[\![E_{\text{arg}}]\!]\texttt{)}\ \ldots\texttt{)}
\end{array}$$

The `primop` transformation is not applicable when O_{prim} is `touch` or when it is a constructor (i.e., `cons`, `vector`, and `cell`). The special case for constructors allows them to hold untouched lazons and eagons.

The straightforward touch introduction scheme sketched above can be optimized to avoid unnecessary touches. For example, it is never necessary to touch a literal, quoted expression, or `lambda`. In fact, it is only necessary to touch expressions that might evaluate to a lazon or eagon. Analysis techniques that conservatively approximate this property could reduce the number of `touch`es introduced.

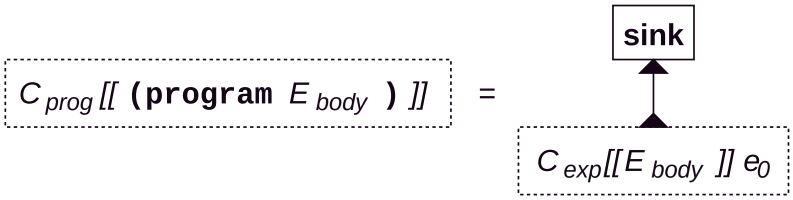
8.3.3 Translating OK to EDGAR

Most of the messy details of compiling OPERA to EDGAR are handled by the OPERA to OK translation described above. The second stage, translating OK to EDGAR, is fairly straightforward. We will describe two compilation functions:

1. $\mathcal{C}_{\text{prog}}$ compiles an OK program to an EDGAR initial graph.

2. $\mathcal{C}_{\text{exp}}$ compiles an OK expression to an EDGAR graph.

$\mathcal{C}_{\text{prog}}$ compiles the OK program `(program` E_{body}`)` to an EDGAR graph rooted at a `sink` node:



$\mathcal{C}_{\text{exp}}$ takes two arguments, an OK expression and an environment, and returns a port of a EDGAR graph. The environment is a function mapping names to EDGAR ports. The empty environment, e_0, maps every name to a distinguished *unbound port* indicating that the name is not bound. The above compilation rule for $\mathcal{C}_{\text{prog}}$ calls $\mathcal{C}_{\text{exp}}$ on the program body and the empty environment.

Compiling simple expressions

$\mathcal{C}_{\text{exp}}$ is a simple recursive descent compiler. Figure 8.18 shows sample clauses for its definition. A literal compiles to a constant node, a quoted symbol compiles to a symbol literal, and `lazon`, `if`, and `primop` expressions compile to `lazon`, `if`, and `primop` nodes. The clauses for `pcall`, `eagon`, `seq1`, `seqn`, and `exclusive` are similarly straightforward, and are not shown.

The only interesting clauses in the definition of $\mathcal{C}_{\text{exp}}$ are for those expressions that manipulate the environment: I, `nex`, `nexrec`, `lambda`, and `pletrec`. When $\mathcal{C}_{\text{exp}}$ reaches a variable reference I, it returns the port associated with I in the environment. If the name is associated with the unbound port, then the expression contains an unbound variable. (Note that such a name can't be a global reference to a primitive procedure since all these have been removed by the eta expansion and procedure in-lining phases of $\mathcal{T}_{\text{exp}}$.)

Compiling `nex` and `nexrec`

The compilation of a `nex` expression extends the environment with the result of recursively compiling the binding expressions, and returns the result of compiling the body expression in the extended environment:

$$\mathcal{C}_{\text{exp}}[\![\texttt{(nex ((}I_1\ E_1\texttt{)}\ \ldots\texttt{(}I_n\ E_n\texttt{))}\ E_{\text{body}}\texttt{)}]\!]e = \mathcal{C}_{\text{exp}}[\![E_{\text{body}}]\!]e[\mathcal{C}_{\text{exp}}[\![E_1]\!]e/I_1, \ldots, \mathcal{C}_{\text{exp}}[\![E_n]\!]e/I_n]$$

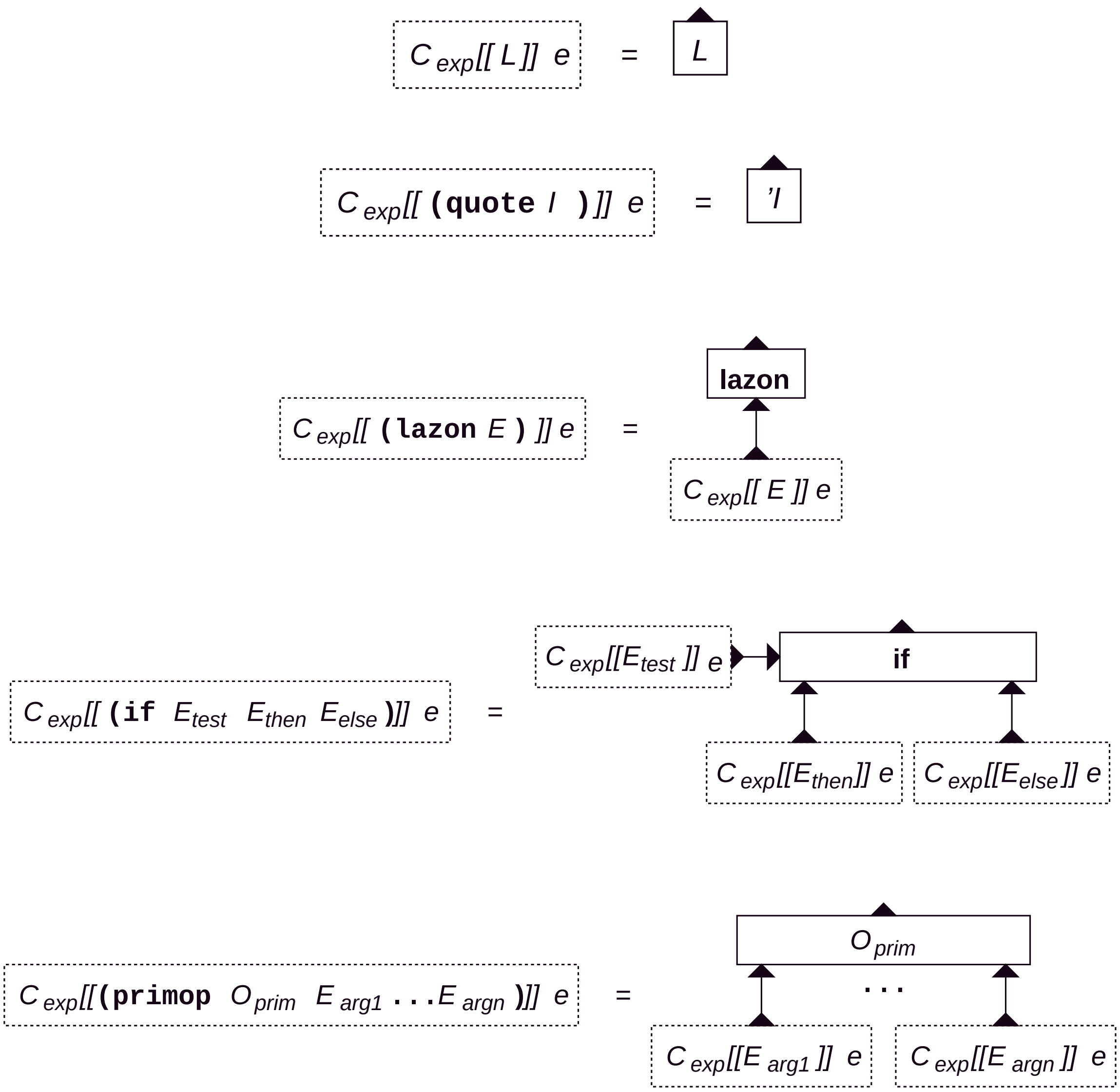


Figure 8.18: Sample clauses for the OK to Edgar compiler.

(The notation $e[\ldots, v_i/I_{\rm i}, \ldots]$ stands for the result of extending environment e with the bindings between identifiers $I_{\rm i}$ and values v_i.) By allowing a subgraph to be named and referred to later by a variable, `nex` permits the creation of directed acyclic graphs.

`Nexrec` permits the full flexibility of graphs by allowing cyclic references through the environment:

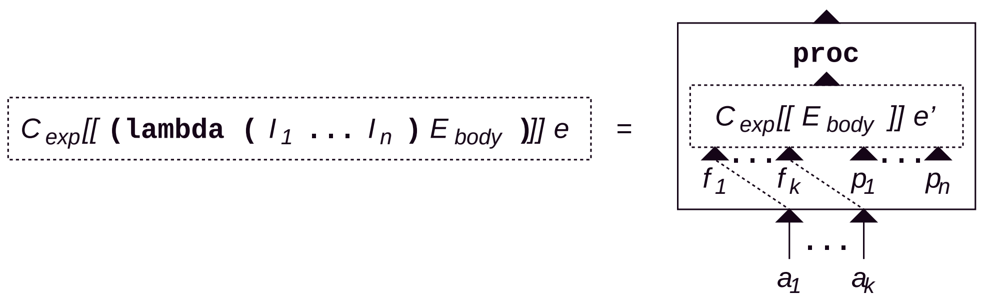
$$\mathcal{C}_{\rm exp}[\![\texttt{(nexrec ((}I_1\ \ E_1\texttt{)}\ \ldots\texttt{(}I_{\rm n}\ \ E_{\rm n}\texttt{))}\ \ E_{\rm body}\texttt{)}]\!]e \ = \mathcal{C}_{\rm exp}[\![E_{\rm body}]\!]e'$$
$$\text{where}\quad e' = e[\mathcal{C}_{\rm exp}[\![E_1]\!]e'/I_1, \ldots, \mathcal{C}_{\rm exp}[\![E_{\rm n}]\!]e'/I_{\rm n}]$$

The environment e' is the fixed point obtained from extending e with bindings between the names $I_{\rm i}$ and the graphs resulting from the compilation of the value expressions $E_{\rm i}$ in e'.

Compiling `lambda`

The compilation of `lambda` expressions depends crucially on the notion of *free variables*. Suppose we view an OK expression as an abstract syntax tree whose leaves are literals, quoted symbols, and variable references, and whose internal nodes are the compound expressions types (e.g., `pcall`, `lambda`, `if`, etc.). Then an occurrence of a variable reference I within an OK expression is *free* if in the corresponding abstract syntax tree there is no I-binding `lambda`, `pletrec`, `nex`, or `nexrec` on the path from the reference to the root of the tree. In other words, I is free if it is not lexically bound by one of the binding constructs. The free variables of E is the set of the names of all the variable references that are free in E.

The compilation of a `lambda` expression in environment e yields a `proc` node whose template results from compiling the body of the `lambda` in an extended environment e':



Suppose that the `lambda` expression being compiled contains k free variables, $J_1 \ldots J_k$. Then $e\,'$is defined as

$$e\,' = e[f_1/J_1, \ldots, f_k/J_k, p_1/I_1, \ldots, p_n/I_n]$$

where f_1, ..., f_k, p_1, ..., p_k are new ports. That is, the extended environment binds each free variable and parameter of the `lambda` expression to a new port. These new ports serve as placeholders for the graph structure that will eventually be the values of the variables.

The `proc` node itself has k input ports which are wired to the results of looking up the free variables J_1, ..., J_n in the environment e. These inputs represent the values of the free variables used within the `lambda` expression. If the free variable was originally bound by a `lambda` or `pletrec`, the result of a lookup will be a placeholder port. However, if the variable was originally bound by a `nex` or `nexrec`, the result of the lookup may be the output port of a fragment of compiled graph structure.

Figure 8.19 shows the result of compiling the following `lambda` expression:

```
(lambda (a)
  (lambda (b)
    (pcall (lambda (c d)
             (* b
                (/ (- c d)
                   (+ c d))))
           (* a a)
           (* b b))))
```

For readability, the parameter ports on the procedure templates have been labelled with the corresponding parameter names. The three nested `proc` nodes in the figure correspond to the three nested `lambda`s of the expression. Note that the template of the innermost `proc` has only one free variable input port because the corresponding `lambda` expression has only one free variable (`b`).

Together, the `lambda` compilation strategy and the EDGAR rewrite rule for `pcall` describe the "plumbing" that allows argument values and the values of free variables to reach their destinations. The input ports to a template within a `proc` node stand for all the values that can potentially be used to compute the body of the template; these include the explicit parameters of the `lambda` as well as implicit parameters — i.e., free variables. A `proc` node holds free variable values as inputs so that they can be supplied as implicit argument values

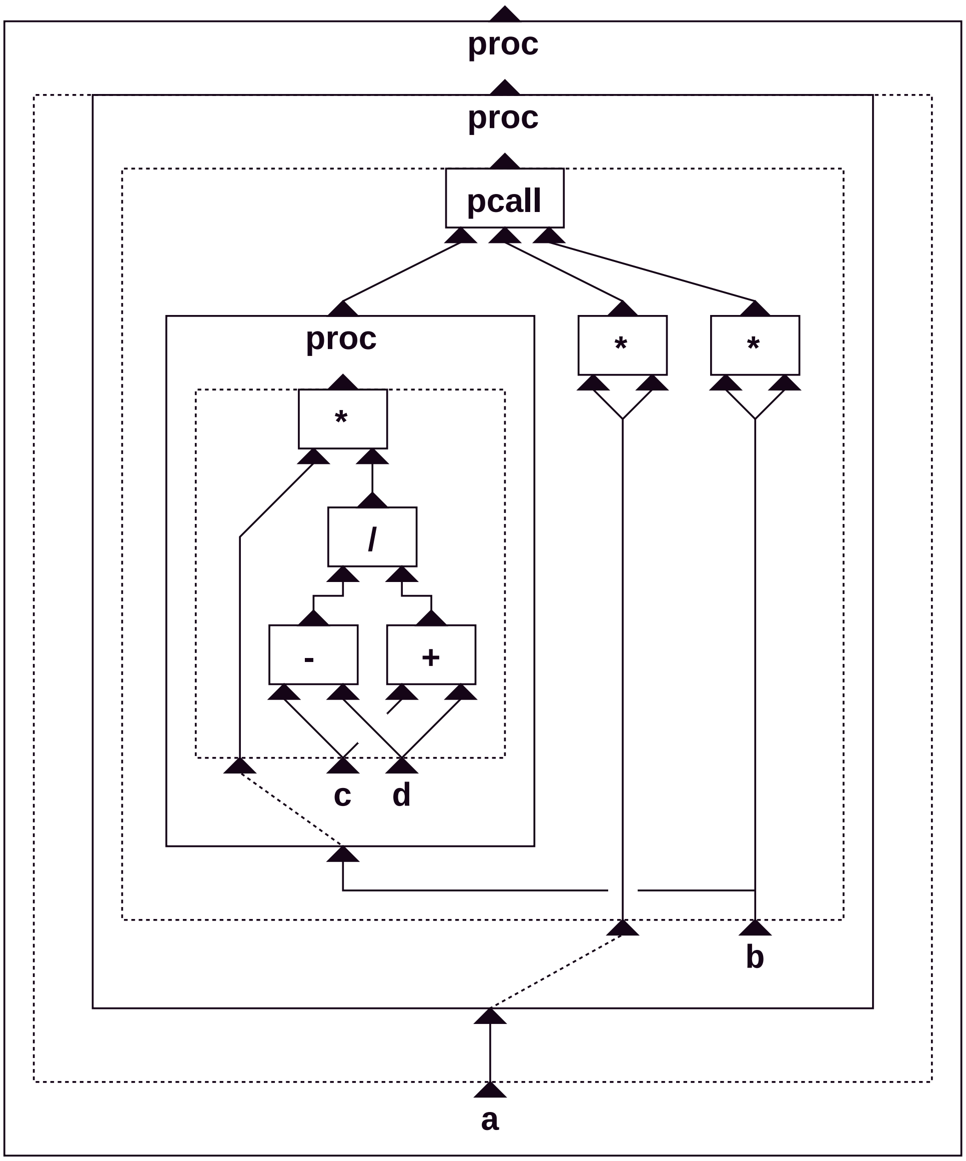


Figure 8.19: Result of compiling a `lambda` expression via $\mathcal{C}_{\text{exp}}$.

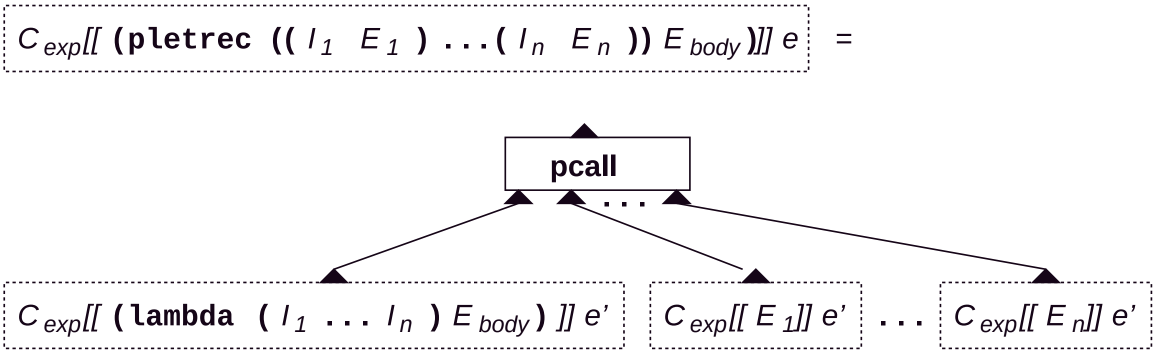
at the point of call. This strategy for managing free variables as implicit arguments is a graphical version of the *lambda lifting* technique for transforming nested procedures into top-level ones [Joh85].

An important detail of the `lambda` compilation process is that the implicit parameters to a procedure are *only* the free variables that appear textually inside of it. A `proc` node only holds onto values that are likely to be used in the computation of the template. In particular, a `proc` node does not hold onto the values of lexical variables that are not referenced by the template. This strategy has desirable consequences for garbage collection. Aggressively dropping references to the values of unused lexical variables means that those values can be garbage collected as soon as they are no longer needed.

In contrast, procedures in the traditional environment model [ASS85] hold onto the values of *all* lexically enclosing variables. As a result, the garbage collection of many provably useless values is delayed because they are spuriously held by a procedure. In programs that manipulate synchrons, the aggressive reference dropping inherent in `lambda` compilation strategy solves the riddle (raised in Section 7.1.3) of how deadlocks implied by the environment model are avoided in OPERA.

Compiling `pletrec`

The compilation of a `pletrec` expression involves the compilation of a `lambda` expression as a subcomponent:



This compilation rule is similar to the desugaring rule for `plet`,

$$\mathcal{D}_{\mathrm{exp}}[\![\texttt{(plet ((}I\ E\texttt{) ...)}\ E_{\mathrm{body}}\texttt{)}]\!] = \texttt{(pcall (lambda (}I\ \texttt{...)}\ \mathcal{D}_{\mathrm{exp}}[\![E_{\mathrm{body}}]\!]\texttt{)}\ \mathcal{D}_{\mathrm{exp}}[\![E]\!]\ \texttt{...)},$$

except that the compilation rule uses an extended environment e' for compiling the subexpressions. The extended environment is the solution of the same fixed point equation used in the compilation of `nexrec`:

$$e' = e[\mathcal{C}_{\text{exp}}[\![E_1]\!]e'/I_1, \ldots, \mathcal{C}_{\text{exp}}[\![E_{\text{n}}]\!]e'/I_{\text{n}}]$$

8.3.4 Notes

We conclude the discussion of OPERA to EDGAR compilation with a few notes:

- *No names:* It is worth emphasizing in the context of OPERA to EDGAR compilation that EDGAR has no names. In fact, one of the most interesting aspects of the compilation process is how it translates all names into graphical connections.

- *Meaning of* OPERA *programs:* OPERA programs are compiled to EDGAR via the function $\mathcal{C}_{\text{prog}} \circ \mathcal{T}_{\text{prog}}$ that composes the two translation stages described above. It is easy to show that the result of this compilation process is an EDGAR initial graph. By the definition of $\mathcal{C}_{\text{prog}}$, it is rooted at a sink; and all of the clauses of $\mathcal{C}_{\text{exp}}$ only create graph structure in which all wires are inactive. We can then define the meaning of an OPERA program as the behavior of the initial graph into which it compiles. The possible values of the OPERA expression can be determined from the outcomes of the behaviors (see Section 8.1.6). In the case where an outcome is an EDGAR value node, it is easy to translate the graph rooted at that node into a legal OPERA value.

  This notion of meaning ignores the fact that the program might read data from or write data to some global entity (such as a file system). In general, the behavior of an OPERA program might depend on global state, and we would be interested in tracking any changes the program makes to the global state. We will assume that global state is handled in one of the ways suggested in Section 8.1.7, and that the notion of program meaning is changed to take into account the initial and final state.

- *Meaning of* OPERA *expressions:* All of the translation machinery described above assumed that we started with an OPERA *program*, not an OPERA *expression*. Yet, in practice, we might be concerned with the meaning of OPERA expressions evaluated by an interactive interpreter (such an interpreter is described in Section 9.1.2). This introduces several complexities. First, it would be necessary to accurately model state maintained by the interpreter (e.g., a global environment that keeps track of bindings introduced by top-level `define`s). Second, it would be necessary to extend the notion of the behavior of an EDGAR graph to express the changes made to the interpreter state by evaluating the graph. This is another example of the global state issue (see Section 8.1.7).

8.4 Alternatives and Extensions

The above discussions of EDGAR and the compilation of OPERA to EDGAR were structured to simplify the presentation. Here we briefly touch upon some extra issues that were ignored above.

- *Primop rules:* There is no real need for EDGAR to represent primitive applications with a special class of nodes. Instead of representing an addition by a special + *application* node, it would be more elegant to handle this situation using the general `pcall` with a special `+` *value* node as its operator. This would reduce the number of rewrite rules and obviate the need for the eta expansion and primitive in-lining phases of the OPERA to EDGAR compiler.

  Why, then, have primitive application nodes? Because without them, the rewrite rules can become more complex. Any rule involving a primitive application (e.g., rules involving `wait`, `touch`, `car`, etc.) would include extra `pcall` nodes. This would make it harder for both people and programs to reason about the rules.

- *Value returns:* The demand driven nature of EDGAR is intended to model the evaluation process of a Scheme interpreter. That is, demand arriving at a node is supposed to correspond to calling `eval` on the expression represented by that node. But the EDGAR rules introduced above fail to accurately model the return of values. For example, in the evaluation of the graph corresponding to `(+ 1 2)`, the newly created result node (`3`) must be demanded before its value is returned. But this implies that the Scheme interpreter calls `eval` on the result of a primitive application, which is definitely not the case.

This behavior can be fixed by dictating that demanded wires connected to the newly created node must immediately enter the returned state. But such a change complicates the primitive application rule. And it also raises new questions. If `(+ 1 2)` is demanded by *two* wires, should the newly created `3` immediately return to both? (I.e., in a concurrent Scheme evaluation model, would it make sense for a value to be returned to two threads at the same time?)

- *Trivial transitions:* In a typical trace, an large percentage of the transitions are trivial ones in which a value node is requested or is returned. These trivial transitions encumber step-by-step reasoning about EDGAR executions, and are a serious annoyance in the program animator based on the EDGAR model (Section 9.1.4).

  It is possible to remove trivial transitions by changing the way that values are handled. An alternative to having value nodes to respond to individual demands is to require that all output wires of every value node are always in the returned state. This alternate strategy is consistent with the interpretation that a returned wire indicates that the node at the source of the wire is a value node. It also corresponds to the common practice in operational semantics of classifying a subset of rewritable expressions as values [Mey, GJ90]. An additional benefit is that it greatly simplifies the handling of `quote`, since a compound quoted expression can be treated as a single value created before the program is executed.[6]

  Unfortunately, the alternate strategy for handling values complicates the EDGAR rewrite rules. At the high level, it invalidates the constraint that a wire cannot enter a returned state until it is first in the demanded state. It also complicates particular rules by increasing the number of cases that must be considered. For instance, it would be necessary to modify the [*pcall-request*] rule to indicate that demand is only propagated to unreturned input wires. New notations could simplify this change;

[6] However, even with the alternate value strategy, it would still be necessary to perform the lifting aspect of the quote removal transformation to get the appropriate sharing of quoted values.

we could introduce a wire pattern that matched only inactive or returned wires, and introduce a meta-rule that demanding a returned wire has no effect. However, such notations would complicate the already detailed rules of EDGAR.[7]

For simplicity, I stuck with the no-return-until-requested model in my presentation of the EDGAR rules above. However, my implementation actually uses the every-value-is-returned strategy.

8.5 Related Work

The development of EDGAR was heavily influenced by previous work in operational semantics and graph reduction. Graph reduction has long been used as an implementation technique for functional programming languages. Turner [Tur79] developed a means for compiling functional programs into graphs of three combinators (S, K, and I) and executing the resulting graphs by graph rewriting. Hughes [Hug82] introduced a method, super-combinators, for compiling a function into a specialized graph rewriting rule. Peyton Jones [Pey87] summarizes numerous issues relevant to graph reduction as an implementation technique.

While EDGAR is clearly a graph reduction framework, it differs in many respects from the work done in the functional programming community. First is a difference in emphasis: EDGAR is primarily a semantic framework, not an implementation technique. The EDGAR rules have been implemented as an interpreter, but only as a proof-of-concept; we make no claim that the EDGAR style of graph reduction is a good implementation technique. In this sense, EDGAR is closer to work on graph rewriting semantics [B+87, AA93].

Because implementation efficiency is not a concern, some of the concepts in EDGAR are cleaner than in other graph reduction frameworks. For example, the ability for EDGAR rules to "move" wires obviates the need for messy special mechanisms like overwriting application nodes and inserting indirection nodes (see [Pey87]). The fact that EDGAR deals with side effects, concurrency, and synchronization clearly separates it from much of the

[7] The added complexity may be due to the fact that using return tokens to represent values may just be a bad idea. The traditional approach of defining a class of value graphs might simplify the presentation of the alternate strategy. This possibility needs to be explored.

other work in the field. Although parallel graph reduction techniques exist [Pey87, Sme93], the focus is again on efficiency, not on semantics. Until relatively recently, side effects have been anathema to the functional programming community, so they rarely appear in graph reduction work (but see [AA93]).

An notable exception to the avoidance of side effects is Bawden's work on connection graphs (CGs) [Baw86, Baw92]. Bawden introduced (CGs) to explore issues of state and linearity in programming languages. Like EDGAR, CGs are a framework in which graph rewrite rules define the semantics of interconnected nodes with labelled ports. Because the systems have different goals, it is not surprising that they differ in many details: CG wires are undirected while EDGAR wires are directed; CGports must be connected to exactly one wire, while EDGAR output ports may be connected to zero or multiple wires; CG rewrite rule patterns may only have two nodes connected by a single wire, while EDGAR rewrite rule patterns may involve any connected set of nodes; CG uses node states to simulate evaluation, while EDGAR uses tokens on wires. Nevertheless, CGs exerted a tremendous influence on the development of OPERA. The notion that procedures hold onto the values of their free variables, the non-deterministic handling of side effects, and many aspects of the OPERA to Scheme compiler were inspired by Bawden's Scheme to CG compiler.

EDGAR's explicit representation of demand was inspired by the demand tokens used in the operational model for Gelernter and Jaganathan's Ideal Software Machine (ISM) [GJ90]. ISM is a model that ascribes sequential and parallel characteristics to dimensions of both space and time. The concurrency allowed by ISM (parallelism in the time dimension) is formally expressed in a semantic framework that combines aspects of Petri nets [Pet77] and graph rewriting. Like Petri nets, the ISM semantics uses tokens to represent concurrency and synchronization; in the case of ISM, the tokens are an explicit representation of demand. As in graph rewriting, the ISM semantics allows graph rewrites to occur where patterns are matched; the patterns usually involve tokens.

Not only was the basic ISM model adopted in EDGAR, but many of EDGAR's rewrite rules (such as those handling procedure calls, conditionals, and sequential evaluation) are almost identical to ones in the ISM semantics (modulo different notation). There are several major differences, however. Foremost is EDGAR's support of synchrons as a new synchronization technology; in fact, the *raison d'etre* of the EDGAR model is to explain synchrons. Another major difference is naming. We find that the ISM model handles naming in a rather ad hoc fashion; by eliminating names altogether, the EDGAR model emphasizes that naming is fundamentally a topological concern (i.e., names express connections). As far as we can determine, the ISM semantics for procedure call is not properly tail recursive; this is an EDGAR feature that is crucial for controlling space behavior. Finally, it appears that ISM's representation of closure environments holds onto values longer than necessary. To be fair, ISM was designed to be a general model for comparing different languages. We view EDGAR as an improved version of ISM targeted at faithfully modelling a wide range of operational issues.

A handful of other systems employ explicit representation of demand. Arvind and Pingali describe a mechanism for simulating demand-driven evaluation in a data flow model; they use data tokens to represent demand [PA85, PA86]. Ashcroft describes a system that combines demand flow (via entities called *questons*) with data flow (via entities called *datons*) [Ash86].

The visual notations used to represent EDGAR graphs were influenced by the notations used in the data flow community (e.g., [Den75, DK82]). We emphasize, however, that the execution models for EDGAR and data flow are fundamentally different. In data flow models, tokens representing values flow through mostly static networks; the same wire can be used transmit many different values. In EDGAR's dynamically-changing networks, a wire may only transmit a single value. Additionally, the demand-driven nature of EDGAR conrasts with the data-driven emphasis of data flow.

The visual notations of EDGAR bear some resemblance to plan calculus representations from the Programmer's Apprentice project [Ric81]. And, in fact, EDGAR was designed to support many of the same kinds of program decompositions considered in that project. EDGAR has the advantage that it is executable, but as a relatively low-level operational model, it does not straightforwardly support the high-level plans at the heart of the Programmer's Apprentice.

It is possible to encode EDGAR's graphical rewrite rules in a more traditional textual fashion. The substitution model described by Meyer [Mey], which was influenced by ideas in

EDGAR, is a step in this direction. Unlike most other text-based operational models, Meyer's model captures the sharing of values in data structures. However, from the viewpoint of pedagogy, we strongly believe that the visual notations provided by EDGAR are vastly superior to any textual mechanism for indicating sharing.

Chapter 9

Experience

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

9.1 Implementation Notes

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

9.1.1 EDGAR

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

Basics of the EDGAR Interpreter

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

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

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

Task Selection Strategies

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

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

Garbage Collection

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

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

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

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

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

Global Cells

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

Metering

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

1. The number of tasks performed during a computation.
2. The number of nodes created during the computation.
3. The maximum number of nodes created during the computation that are in use at any intermediate point of the computation.
4. The number of nodes accessible at the end of a computation.
5. The process time required by a computation.
6. The real (wall clock) time required by a computation.

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

Debugging

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

Liberties

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

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

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

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

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

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

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

9.1.2 OPERA

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

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

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

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

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

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

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

9.1.3 SYNAPSE

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

9.1.4 The DYNAMATOR

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

Snaphsot Representation

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

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

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

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

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

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

Dynamics

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

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

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

Graph Layout

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

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

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

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

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

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

Additional Features

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

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

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

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

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

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

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

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

9.2 Testing

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

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

9.2.1 Outcomes

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

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

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

9.2.2 Computations

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

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

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

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

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

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

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

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

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

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

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

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

9.2.3 Space Requirements

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

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

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

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

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

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

Average

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

```
;;; Monolithic version

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


;;; Modular version

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

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

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

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

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

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

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

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

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

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

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

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

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

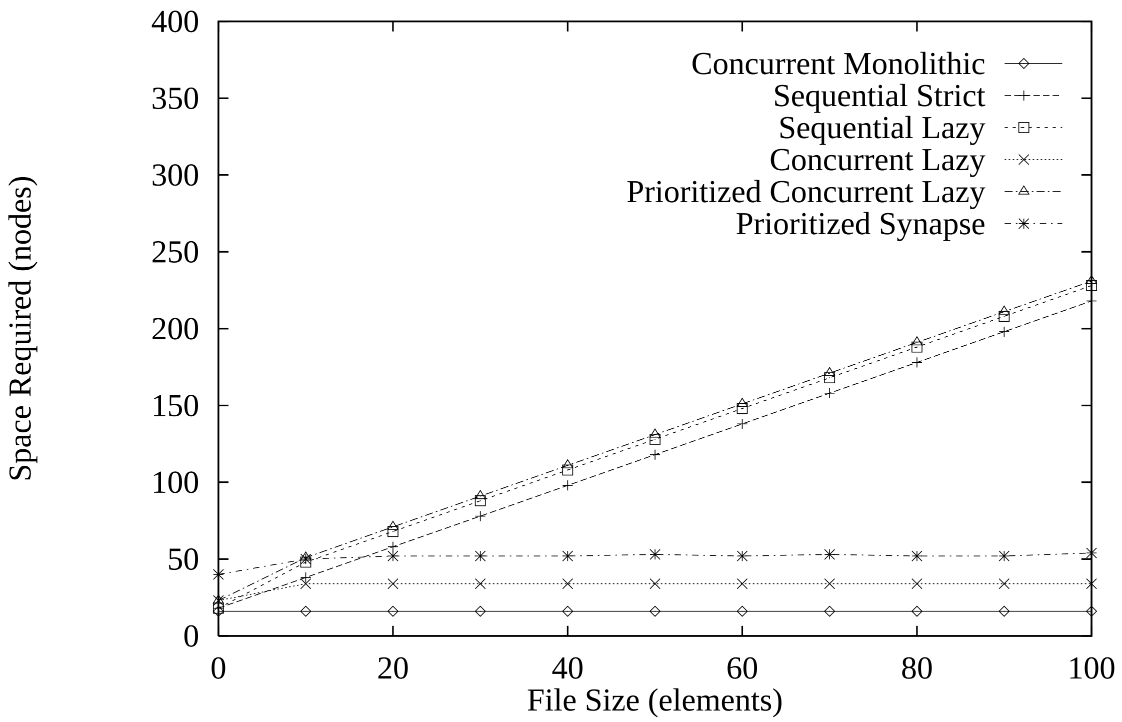


Figure 9.2: Comparison of the space requirements of various averaging programs as a function of file size.

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

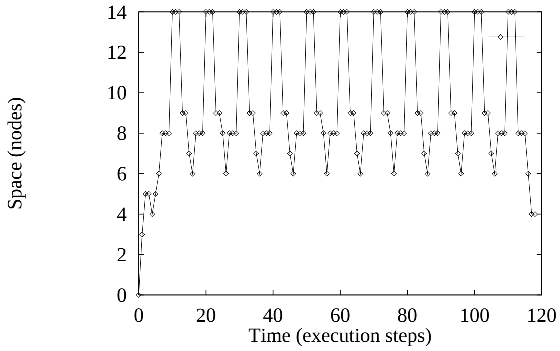
1. *Flat*: The three constant-space computations (concurrent monolithic, concurrent lazy, and prioritized SYNAPSE) all exhibit a broad flat band in which the space usage alternates between high and low values ten times. Each high value is the size of the graph soon after the beginning of a new iteration,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

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

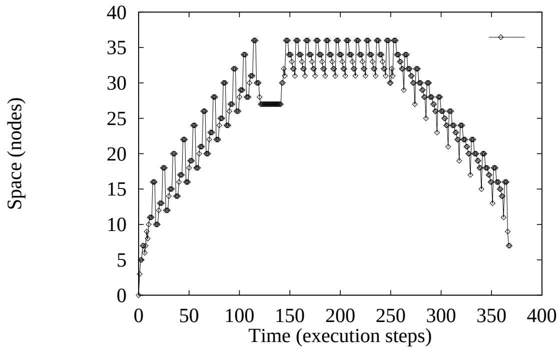
2. *Peak*: The sequential lazy and prioritized concurrent lazy programs exhibit profiles that gradually rise to a peak and then fall off more rapidly. The rise is due to the fact that each element read from the file is buffered for later use; the fall occurs in the phase of the computation that processes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

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

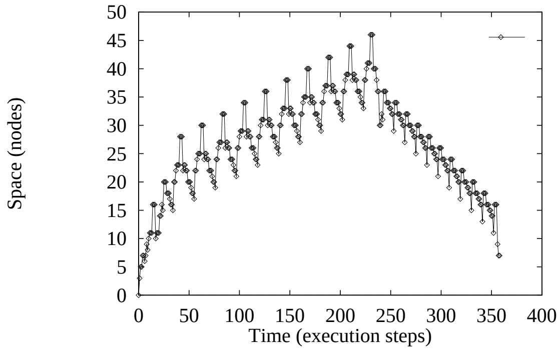
[7]The SYNAPSE profile was generated in an unfiltered implementation of SYNAPSE The time overhead would be even higher in the filtered version.



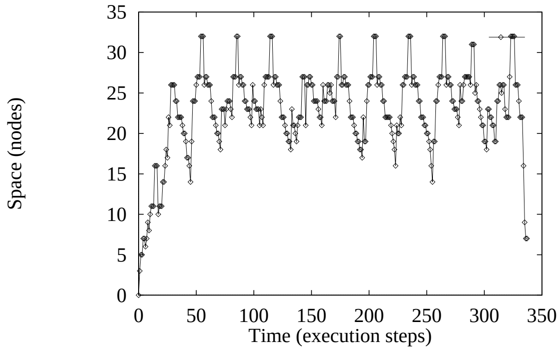
(a) Concurrent monolothic



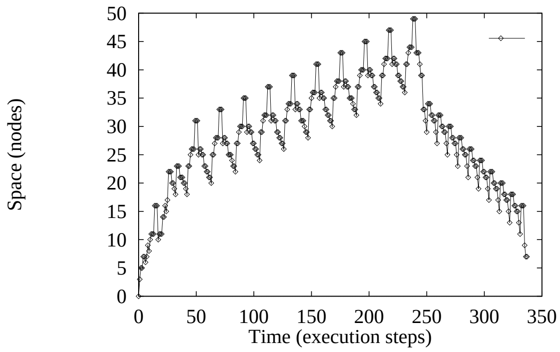
(b) Sequential strict



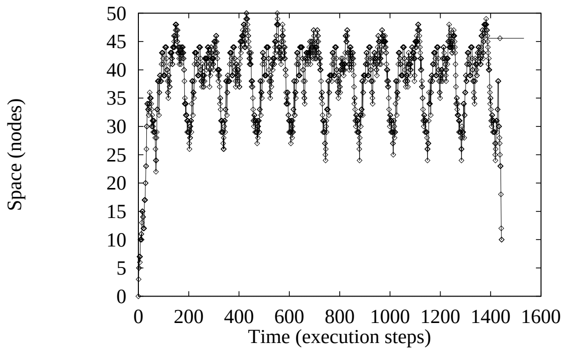
(c) Sequential lazy



(d) Concurrent lazy



(e) Prioritized concurrent lazy



(f) Prioritized SYNAPSE

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

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

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

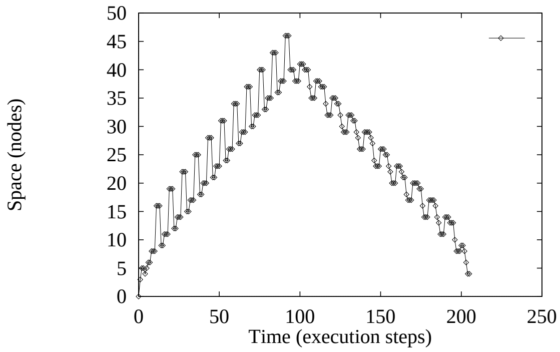
```
(define (average-file-up:mono filename)
  (let ((port (open-input-file filename)))
    (define (recur)
      (let ((elt (read port)))
        (if (eof-object? elt)
            (begin (close-input-port port)
                   (cons 0 0))
            (let ((sum&count (recur)))
              (cons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

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

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

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

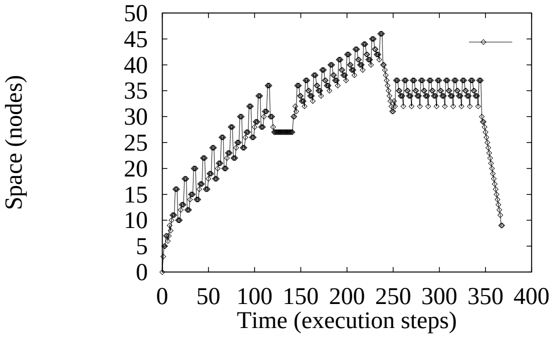
expected, all versions require space linear in the size of the file due to the stacked operations and their arguments (details omitted).

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

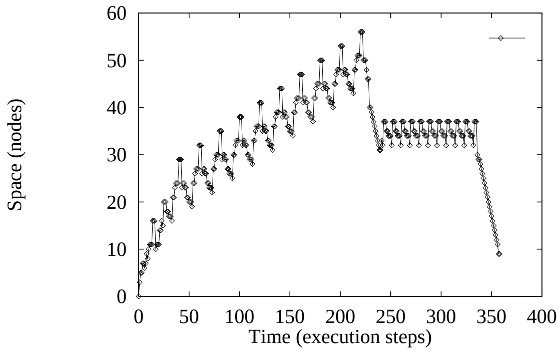
[8] More precisely, the rise is due to the creation of a stack of pending `cons` operations. This is followed by a brief stack-popping phase in which the pending `cons` operations are executed to create a list. This stack-popping phase appears as a short straight line segment in the profile at the beginning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



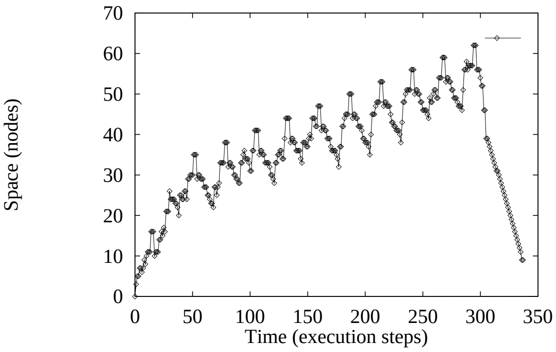
(a) Concurrent monolothic



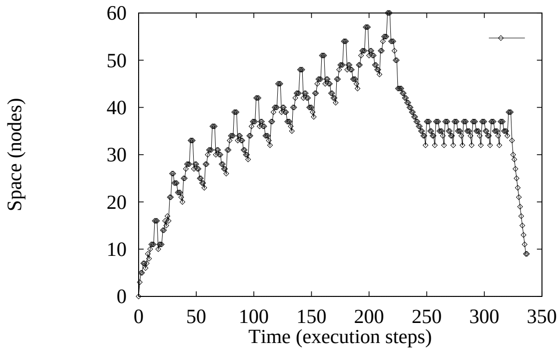
(b) Sequential strict



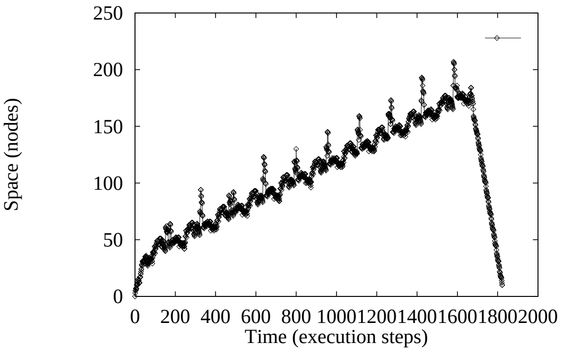
(c) Sequential lazy



(d) Concurrent lazy



(e) Prioritized concurrent lazy



(f) Prioritized SYNAPSE

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

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

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

Alpha Rename

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

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

where $body_i$ is recursively defined by

$$body_0 = \texttt{x}$$

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

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

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

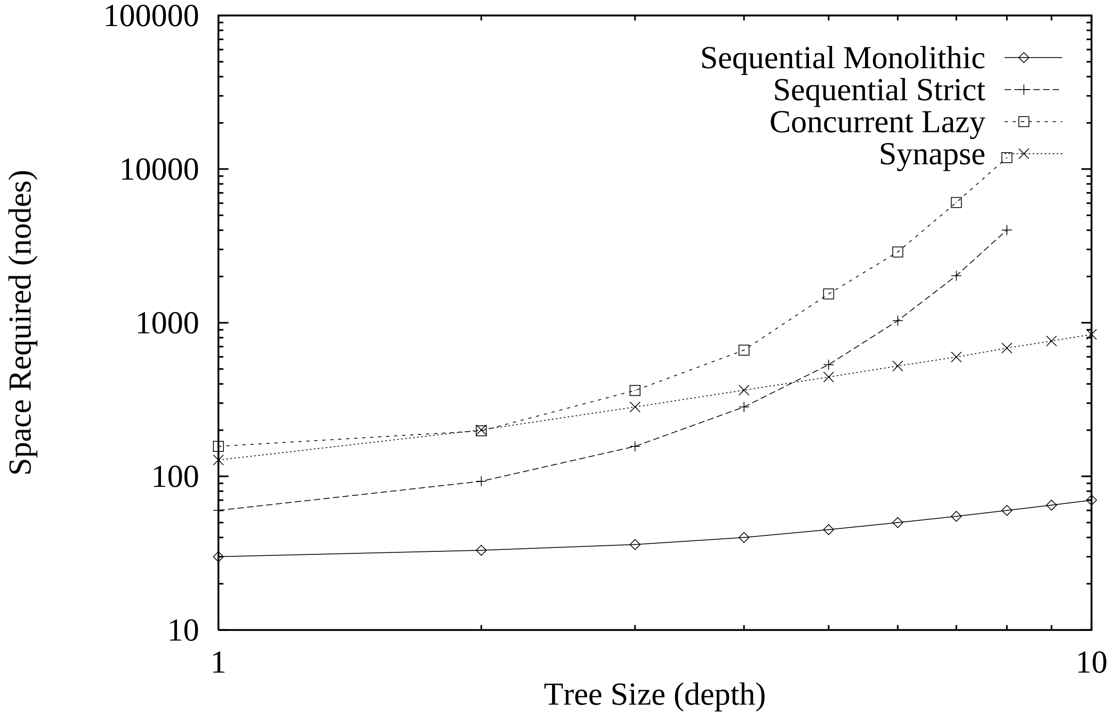


Figure 9.6: Log-log plot of the space consumed by various versions of the modified alpha renaming program as a function of the depth of the input tree.

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

9.3 Lessons

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

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

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

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

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

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

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

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

3. *Laziness requires defensive programming strategies*:

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

Consider the following two lambda expressions written in OPERA:

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

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

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

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

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

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

4. *Environment models are problematic*:

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

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

Chapter 10

Conclusion

10.1 Summary

The solutions to many programming problems can naturally be expressed as the composition of procedures that manipulate lists and trees. By capturing common idioms in reusable pieces, this modular program organization facilitates reading, writing, modifying, and reasoning about programs. Unfortunately, issues of space, time, and operation scheduling often compel programmers to manually interweave several idioms into a single complex loop or recursion. The result is a monolithic program that is significantly harder to construct, understand, and extend than the modular version.

I have developed a technique that enables programmers to achieve the desirable space characteristics and operation scheduling of intricate loops and recursions in modular list and tree programs. The technique is based on a lock step processing model that allows networks of aggregate data operators called *slivers* to capture the fine-grained operational behavior of monolithic procedures. Slivers communicate via *synchronized lazy aggregates*, standard interfaces that transmit not only elements but control information that specifies when the elements are computed. Control information is in the form of *synchrons*, a novel barrier synchronization mechanism that is fundamentally tied to automatic storage management. Synchrons enable the barriers represented by strict monolithic procedures call to be simulated by a network of concurrently executing slivers.

The sliver technique combines the simplicity and expressiveness of lazy data with the synchronization and storage efficiency of interprocess communication channels. The sliver approach is a dynamic version of Waters's series technique that is able to express general linear and tree-shaped recursive computations in addition to iterations. By allowing the complexities of synchronization to be hidden from the programmer, slivers improve upon the concurrency and synchronization techniques suggested by Hughes for controlling space behavior in modular programs.

I have implemented a tower of languages (EDGAR, OPERA, and SYNAPSE) that support the lock step processing model. EDGAR is an explicit demand graph reduction model in which the operational details of concurrency, synchronization, and non-strictness are formalized. OPERA is a dialect of Scheme in which these operational features are embedded. SYNAPSE is a collection of higher order OPERA procedures that manipulate synchronized lazy lists and trees. I have shown how a number of intricate computations can naturally be expressed as modular combinations of SYNAPSE procedures, and that the resulting programs mimic the detailed space behavior and operation scheduling of hand-crafted loops and recursions. Synchronized lazy aggregates can also be used in conjunction with conventional strict and lazy aggregates to partition a computation into loosely coupled networks of tightly coupled components.

The slivers technique addresses the space overheads, but not the time overheads, associated with aggregate data programs. While sliver programs have the same time complexity as corresponding monolithic programs, prototype implementations indicate that they suffer significant time overheads due to the manipulation of intermediate data and the management of fine-grained concurrency and synchronization. Nevertheless, I expect that it will be possible to eliminate these overheads in many cases by employing techniques similar to those used by Waters to compile series programs into monolithic loops and recursions.

Despite the pragmatic drawbacks, the work described here represents an important step towards reducing the tension between modularity and control in programming. At the very least, slivers and synchronized lazy aggregates constitute an executable specification language for expressing operational characteristics of programs. More importantly, the notions of computational shape and lock step processing developed here provide new ways for programmers to think and talk about computation.

10.2 Contributions

Here is a summary of the major contributions of this work:

- I have presented a detailed analysis of why existing modularity techniques fail to preserve important operational characteristics of monolithic loops and recursions.

- I have laid the groundwork for a theory of *computational shape* that describes how the operational processing patterns of a computation can be composed from the patterns of its components. The preliminary notion shape presented here successfully captures intuitions about linear and tree-structured computations.

- I have developed a lock step model of computation that allows programs written in the aggregate data style to control space behavior and operation scheduling in a modular fashion. The model introduces two new abstractions that successfully encapsulate the details of lock step processing: a *sliver* is a processing module and a *synchronized lazy aggregate* is a data structure that guarantees the lock step processing of the slivers that produce and consume it. I have demonstrated the utility of the lock step processing model in the context of SYNAPSE, a list and tree processing language.

- I have invented a novel synchronization mechanism, the *synchron*, that interacts with automatic storage management to provide a general form of barrier synchronization. Synchrons are the key technology for implementing synchronized lazy aggregates and achieving the lock step processing of slivers.

- I have designed a semantic framework, *explicit demand graph reduction* (EDGAR), that models concurrency, synchronization, non-strictness, and side effects in an intuitive way. I have implemented a graph-based interpreter based on EDGAR and used it as a foundation for OPERA, a concurrent dialect of Scheme that supports synchrons.

- I have implemented a graphical program animator, the DYNAMATOR, that visually illustrates the step-by-step execution of the EDGAR model. The DYNAMATOR shows great promise as a tool for debugging and pedagogy.

10.3 Future Work

10.3.1 Expressiveness

The SYNAPSE implementation validates the lock step model upon which slivers are based, but it is far from an ideal system for expressing computations in a modular way. While it is fairly easy to express a wide range of linear computations in terms of existing synquence slivers, there is still much functionality missing. More linear primitives, along the lines of those developed for series [Wat90], need to be added.

Syndrites are in a considerably more embryonic state than synquences. The current set of syndrite slivers is very weak. The mechanisms for specifying different orders and directions of sequential tree traversals are rather ad hoc; better abstractions for sequential processing of trees need to be found. It would also be helpful to develop a sliver that sequentializes parallel tree manipulations in a reusable way. Designing slivers whose processing shapes are dynamically determined rather than being statically defined is another important avenue of exploration. For example, imagine a syndrite that performs a left-to-right pre-order walk if its argument is even, but a right-to-left post-order walk if its argument is odd.

To drive the design of an expressive suite of syndrite slivers, the sliver technique needs to be tested out on a wide range of complex tree-shaped computations. I have only experimented with a few relatively simple tree programs: free and bound variable analysis, alpha renaming, lambda lifting, and pattern matching. At the very least, slivers should be used to duplicate results from attribute grammar research within the aggregate data style. A more ambitious undertaking would be a detailed study of existing tree programs along the lines of Waters's study of iterative programs [Wat78, Wat79].

Compilers and interpreters are particularly relevant candidates for study. One of the main pragmatic motivations for slivers was to be able to describe compilers and interpreters in a modular, but still efficient, fashion. Currently, interpreters are not easily accomodated by SYNAPSE because the dynamic evaluation trees generated by an interpreter tend to exhibit complex dependencies between subtrees. For example, when evaluating an application node in a Scheme interpreter, the environment associated with the result of evaluating the operator subtree is needed in the subtree for evaluating body; this dependency does not correspond to an existing syndrite sliver.

Filtering is an area that requires more work. While reusable filtering works surprisingly well for synquences, the details of syndrite filtering have yet to be ironed out. Compacting the elements of a filtered list can often be accomplished using the loose coupling afforded by stream buffers; but this technique can lead to deadlock in situations with fan-out. The default gap handling properties of mappers and accumulators is often inappropriate when multiple inputs are involved. The existence of a large corpus of tree examples would help in the design of better filtering constructs.

Cyclic dependencies are another area requiring further investigation. Some problems naturally decompose into components that exhibit cyclic dependencies. For example, in the FX language [GJSO92], name resolution and type reconstruction are conceptually distinct stages of the implementation that happen to depend on each other. In practice, the implementations of these stages must be manually interwoven. Slivers offer the possibility that this process could be expressed as a cyclic combination of a name resolution sliver and a type reconstruction sliver. While I have successfully implemented a cyclic Fibonacci network based on slivers,[1] I have not yet developed robust enough abstractions for experimenting with more complex examples.

Tail calls should be expressed in a more elegant manner. The tail call of a monolithic procedure is currently simulated in SYNAPSE slivers by using a special calling mechanism that does not rendezvous at an up synchron. But this requires two versions of every calling abstraction. Is it possible to design a cleverer calling convention that automatically waits on an up synchron only if there is pending work to be done?

10.3.2 Pragmatics

In order for synchronized lazy aggregates to be a practical technique, it is necessary to significantly reduce the time and space overheads exhibited by the current SYNAPSE and OPERA implementations. This area is ripe for exploration.

The time overhead is by far the biggest problem. While SYNAPSE programs have the

[1] It turns out that there can never be a cycle consisting purely of slags within a sliver network. At least one connection in every cyclic dependency must be a non-slag.

same order of growth in time as the monolithic programs that they model, preliminary results indicate that they take more than an order of magnitude more execution steps than monolithic programs executed under the same graph reduction model. The extra steps are due to the following sources:

- Packaging and unpackaging the elements of slag.
- Manipulating and rendezvousing on synchrons.
- Manipulations for handling filtering appropriately.
- Handling of `lazon`s, `eagon`s, and touches that control fine-grained behavior.

Because the comparisons were performed in a graph reduction model, these overheads do not include any overheads of the graph reduction model itself. In practice, straightforward implementations of the fine-grained concurrency and reference-counting garbage collection implied by EDGAR and OPERA are likely to incur additional significant time penalties compared to traditional sequential implementations of monolithic programs.

While SYNAPSE programs can exhibit the same order of growth in space as a monolithic program, they still have higher constants due to the memory requirements of slags. For example, a constant-space iteration composed out of two SYNAPSE slivers requires extra storage for the intermediate slag structure necessary to communicate a single element from one sliver to another.

There are two basic approaches for reducing the overheads associated with SYNAPSE

1. The *dynamic approach* is to develop efficient mechanisms for handling the fine-grained concurrency, synchron synchronization, and non-strictness required by OPERA.
2. The *static approach* is to develop static analysis and compilation techniques that remove the sources of overhead.

Unfortunately, the dynamic approach does not address many of the sources of SYNAPSE inefficiency. In particular, it does nothing to reduce the allocations, stores, and reads from intermediate data structures, nor does it remove the extra manipulations associated with filtering. Nevertheless, if other overheads are brought down to low enough levels, these remaining overheads may be deemed acceptable for certain classes of programs. Since programmer time is often more valuable than machine time, the ease with which a program can be written can be more important than how fast it is.

The static approach holds more promise for increased efficiency. Ideally, it should be possible to eliminate the need for run-time overhead completely by compiling many sliver networks into efficient monolithic recursive procedures. Techniques that are particularly worth exploring in this context include series compilation [Wat91], deforestation [Wad88], partial evaluation [WCRS91], and attribute grammar evaluation [DJL88]. Series compilation is based on representing a series operator as collection of code fragments that characterize different aspects of the operator; compilation consists of gluing together corresponding fragments for a network of series operators. This approach appears promising for slivers, but the details need to be worked out. Both deforestation and partial evaluation techniques can have trouble dealing with general recursions, but the limited forms of recursions in SYNAPSE and the explicit synchronization information available may provide enough constraints to make the techniques applicable.

It is worth emphasizing that requiring slivers networks to be compiled into monolithic recursive procedures will result in reduced expressive power. Since any such compilation technique would need to determine the structure of a sliver network at compile time, many of the more expressive aspects of slivers (e.g., higher-order slivers, repeated application of a sliver, conditionally choosing a sliver at runtime) would have to be curtailed. Of course, an option would be to compile only those networks that are statically determinable, and leave the more expressive cases to be handled by existing dynamic methods. But if the goal is to eliminate the run-time overheads of features like fine-grained concurrency, this option is not viable.

If dynamic overheads are deemed allowable, then other forms of static analysis (in addition to the compilation techniques mentioned above) can be used to reduce these overheads. For example, using the technique of *effect analysis* [LG88], it may be possible to distinguish expressions that need to be evaluated concurrently from those for which sequential evaluation is sufficient.

10.3.3 Computational Shape

The notions of computational shape introduced in Chapters 4 and 5 are alluring but very preliminary and informal. An important area of future research is to formalize these notions into a *shape calculus* that can be used as a basis to help both humans and machines reason about programs. A shape calculus would capture the essential interfaces, call structures, and internal dependencies of process fragments like slivers and use this information to determine both pairwise sliver compatibility and the processing shapes of lock step components.

A shape calculus could be the basis for many practical tools. In analogy with type systems, it is possible to imagine a *shape system* in which the shape of sliver network can be determined from the shape of its components by *shape reconstruction*. Shape analysis would be helpful for conservatively predicting whether a sliver network could deadlock. Any sliver compilation technique would presumably have to employ some sort of shape analysis.

Shape-based reasoning shows promise as a new way to approach program transformations. Existing algebraic techniques may benefit by incorporating a notion of shape. Based on a library of slivers partitioned according to function, it may be possible to improve the operational characteristics of a given program by replacing some slivers with differently-shaped, but functionally equivalent, slivers.

Another direction to explore is extending the simple linear and tree shapes suggested by this research. Are there processes shaped like arrays, DAGs, and graphs? What process shapes are suitable for describing data parallel computations? How can the process shapes generated by fancy control constructs (e.g., non-local exits and continuations) be expressed? Is it possible to convert a process of one shape to one of another shape? (For example, perhaps sequential tree walkers can be described as linear processes that are appropriately "bent" to form a tree.)

It would also be worthwhile to explore synergies between the notion of shape developed here and other notions of data, communication, and computation shapes. The dependency analysis underlying many attribute grammar systems [DJL88] is relevant to the design of the shape calculus. It may well turn out that the shape calculus amounts to a modular version of this analysis. Jay and others are currently developing a theory of "shapely types" that factors data structures into their shapes and their elements. For example, the same sequence of elements can be stored in a list, an array, or as the leaves of different kinds of trees; a shapely type describes the structure holding onto the sequence [Jay]. Size and access inference for data parallel programs [CBF91] is another line of work that strongly suggests a notion of process shapes. In his work on paralations [Sab88], Sabot described an intriguing notion of communication shape.

10.3.4 Synchronization

Synchrons are a powerful synchronization mechanism that may have many other uses beyond their role in synchronized lazy aggregates. For example, it might be possible to use them as the basis for a general temporal constraint solver. The time-based nature of synchrons suggests a *temporal calculus* in which actions are specified to happen before, after, or between synchrons. Early versions of the OPERA supported `before`, `after`, and `between` constructs on synchrons, but they were ultimately abandoned because their semantics was never satisfactory. Working out the details of a temporal calculus would be an interesting project.

Synchrons are currently implemented using reference counting garbage collection. Reference counting, which requires explicit deallocation of every object, has a high overhead compared to techniques (like stop and copy) that only examine live data. Is it possible to implement synchrons more efficiently?

Synchrons are an example of a growing number of data structures that interact with garbage collection. Other examples include T's populations [RAM83], MultiScheme's pairs and finalization objects [Mil87], and Hughes's `synch` construct [Hug83, Hug84]. This trend raises a number of questions: Are there other examples of these structures waiting to be uncovered? What is an appropriate set of primitives and means of combination for constructing new ones? What are efficient techniques for implementing these structures? What are the appropriate formalisms for describing their semantics?

10.3.5 Theoretical Directions

A great deal of work can be done in formalizing the concepts introduced informally in this document. In addition to the shape calculus mentioned above, EDGAR is fertile ground for some theoretical explorations. For example:

- *Confluence*: It would be nice to know under what conditions the model is confluent (i.e., has the property that all computations from the same initial graph end at the same final graph).

- *Sliver Correctness*: Many of the sliver implementations (especially scanners) are extremely complex. EDGAR provides a formal model for their execution; can it help to prove them correct?

- *Behavioral Equivalence*: I have claimed that sliver networks are operationally equivalent to monolithic programs “modulo management operations”. Classical approaches to process equivalence (like CSP's trace equivalence [Hoa85]) appear too weak to handle the crucial notion of equivalent space behavior. EDGAR seems to be a natural starting point for a richer notion process equivalence that explicitly models space consumption. The hard part is defining what “modulo management operations” means in this context. Pomset models of concurrent computation [Pra86, HA87] may be helpful for defining an EDGAR-based notion of process equivalence.

10.3.6 Pedagogy

The computational models and tools developed for this work have important pedagogical applications. The EDGAR model is an extremely rich computational model in which a wide variety of programming issues can be studied: iteration, linear and tree recursion, tail calls, first-class procedures, procedure calling mechanisms, laziness, eagerness, side effects, environment structures, continuations, concurrency, synchronization, and nondeterminism. The fact that all of these issues can be explored through the graphical animation of the DYNAMATOR makes EDGAR a natural choice for illustrating the dimensions of these issues.

The notions of shape and lock step processing presented in this dissertation suggest new ways for teaching the structure of both modular and monolithic computations. For example, the difference between linear iterations and recursions can be introduced by means of sliver diagrams annotated with shapes. The fact that elements between a *down* process and an *up* process form an explicit stack helps to convey the essence of recursion. Shapes can aid in the discussion of program organization and transformation. And the fact that slivers are mix-and-match pieces emphasizes the principles of modularity.

Slivers and shapes not only open up new teaching avenues, but they are a potentially powerful language for helping programmers express processes to each other and to their computers. It remains to be seen whether their potential will be realized.

Bibliography

[AA93] Zena Ariola and Arvind. Graph rewriting systems: Capturing sharing of computation in language implementations. Computation Structure Group Memo 347, MIT Laboratory for Computer Science, April 1993.

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

[Ame89] American National Standards Institute. *American National Standard for Information Systems Programming Language Fortran: S8(X3.9-198x)*, 1989.

[AN89] Arvind and Rishiyur S. Nikhil. A dataflow approach to general-purpose parallel computing. Computation Structure Group Memo 302, MIT Laboratory for Computer Science, July 1989.

[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.

[Ash86] E. A. Ashcroft. Dataflow and eduction: Data-driven and demand-driven distributed computing. In J. W. deBakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency: Overviews and Tutorials*, pages 1–50. Springer-Verlag, 1986. Lecture Notes in Computer Science, Number 224.

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

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

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

[B$^+$87] H.P. Barendregt et al. Toward an intermediate language based on graph rewriting. In *PARLE: Parallel Architectures and Languages Europe, Volume 2*, pages 159 – 175. Springer-Verlag, 1987. Lecture Notes in Computer Science, Number 259.

[Bac78] John Backus. Can programming be liberated from the von Neuman style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):245–264, August 1978.

[Bar84] H.P Barendregt. *The Lambda-calculus: Its Syntax and Semantics*. North-Holland, 1984.

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

[Baw86] Alan Bawden. Connection graphs. In *Symposium on Lisp and Functional Programming*, pages 258–265. ACM, August 1986.

[Baw92] Alan Bawden. *Linear Graph Reduction: Confronting the Cost of Naming*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1992.

[Bel86] Francoise Bellegarde. Rewriting systems on FP expressions that reduce the number of sequences yielded. *Science of Computer Programming*, 6(1):11–34, January 1986.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, pages 239–250, 1984.

[Bir86] Richard S. Bird. An introduction to the theory of lists. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design (NATO ASI Series, Vol. F36)*, pages 5–42. Springer-Verlag, 1986.

[Bir88] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science (NATO ASI Series, Vol. F55)*, pages 5–42. Springer-Verlag, 1988.

[Bir89a] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, pages 122–126, 1989.

[Bir89b] Andrew Birrel. An introduction to programming with threads. SRC Report 35, Digital Equipment Corporation, January 1989.

[BL93] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *25th Annual ACM Symposium on Theory of Computing*. ACM, May 1993.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Ble92] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, Carnegie-Mellon University Computer Science Department, January 1992.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[CBF91] Siddhartha Chatterjee, Guy E. Blelloch, and Allan Fisher. Size and access inference for data-parallel programs. In *Programming Language Design and Implementation '91*, pages 130–144. ACM, 1991.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In *Symposium on Lisp and Functional Programming*, pages 11–20. ACM, 1992.

[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.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

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

[Dar82] John Darlington. Program transformation. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. 1982.

[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.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages (NATO Advanced Study Institute)*, pages 43–112. London: Academic Press, 1968.

[DJL88] Pierre Deransart, Martin Jordan, and Bernard Lorho. *Atribute Grammars*. Springer-Verlag, 1988. Lecture Notes in Computer Science, Number 323.

[DK82] Alan L. Davis and Robert M. Keller. Data flow program graphs. *IEEE Computer*, pages 26–41, February 1982.

[DR76] John Darlington and R.M.Burstall. A system which automatically improves programs. *Acta Informatica*, pages 41–60, 1976.

[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.

[For91] Alessandro Forin. Futures. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 219–241. MIT Press, 1991.

[GJ90] David Gelernter and Suresh Jagannathan. *Programming Linguistics.* MIT Press, 1990.

[GJSO92] David Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Report on the FX-91 programming language. Technical Report MIT/LCS/TR-531, MIT Laboratory for Computer Science, February 1992.

[GLJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional Programming and Computer Architecture*, 1993.

[GM84] Richard P. Gabriel and John McCarthy. Queue-based multi-processing Lisp. In *Symposium on Lisp and Functional Programming*, pages 25–44. ACM, August 1984.

[GW78] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *Conference Record of the Fifth ACM Conference on the Principles of Programming Languages*, pages 1–8, 1978.

[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.

[Hen80] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall, 1980.

[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.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

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

[HS86a] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12), 1986.

[HS86b] Paul Hudak and Lauren Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 243–254, January 1986.

[Hug82] R. J. M. Hughes. Super-combinators: A new implementation technique for applicative languages. In *Symposium on Lisp and Functional Programming*, pages 1–10, August 1982.

[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.

[Hug85] R. J. M. Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*, pages 129–146, 1985. Lecture Notes in Computer Science, Number 201.

[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.

[Jay] C. Barry Jay. Personal correspondence.

[JG89] Pierre Jouvelot and David K. Gifford. Communication effects for message-based concurrency. Technical Report MIT/LCS/TM-386, MIT Laboratory for Computer Science, February 1989.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203, September 1985. Lecture Notes in Computer Science, Number 201.

[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.

[K+86] David Kranz et al. Orbit: An optimizing compiler for Scheme. In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986.

[Kat84] Takuya Katayama. Translation of attribute grammars into procedures. *Transactions on Programming Languages and Systems*, 6(3):345–369, 1984.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu73] Donald E. Knuth. *The Art of Computer Programming. 2nd ed. Vol. 1: Fundamental algorithms.* Addison-Wesley, 1973.

[Kos84] Kai Koskimes. A specification language for one-pass semantic analysis. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 179–189, Tama-City, Tokyo, June 1984.

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

[KW92] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. Technical Report CU-CS-613-92, University of Colorado at Boulder, September 1992.

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

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 47–57, 1988.

[Mey] Albert R. Meyer. A substitution model for Scheme: Formal definitions. In preparation.

[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.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[PA85] Keshav Pingali and Arvind. Efficient demand-driven evaluation (I). *ACM Transactions on Programming Languages and Systems*, 7(2):311–333, April 1985.

[PA86] Keshav Pingali and Arvind. Efficient demand-driven evaluation (II). *ACM Transactions on Programming Languages and Systems*, 8(1):109–139, January 1986.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–250, 1977.

[Pet84] Alberto Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *ACM Conference on Lisp and Functional Programming*, pages 273–281, 1984.

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

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, September 1981.

[Pra86] V. R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[RAM83] Jonathan Rees, Norman I. Adams, and James R. Meehan. The T manual (third edition). Technical report, Yale University Department of Computer Science, March 1983.

[Ric81] Charles Rich. Inspection methods in programming. Technical Report AI-TR-604, MIT Artificial Intelligence Laboratory, June 1981.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RS87] J. R. Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing, Vol 2*, pages 2–16, 1987.

[RW90] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. Addison-Wesley, 1990.

[Sab88] Gary W. Sabot. *The Paralation Model*. MIT Press, 1988.

[Sme93] Jacobus Edith Willem (Sjaak) Smetsers. *Graph Rewriting and Functional Languages*. PhD thesis, Katholieke Universiteit Nijmegan, February 1993.

[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.

[Ste90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 1990.

[Str71] H. R. Strong. Translating recursion equations into flow charts. *Journal of Computer and System Sciences*, 5:254–285, 1971.

[Tra88] Kenneth R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, MIT Laboratory for Computer Science, October 1988.

[Tur79] D. A. Turner. A new implementation for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pages 1–16, 1985. Lecture Notes in Computer Science, Number 201.

[Tur94] Franklyn Turbak. Slivers: Computational modularity via synchronized lazy aggregates. Technical Report AI-TR-1466, MIT Artificial Intelligence Laboratory, 1994. Revised version of doctoral dissertation. In prepartion.

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

[Wad84] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *ACM Symposium On Lisp and Functional Programming*, pages 45–52, 1984.

[Wad85] Philip Wadler. Listlessness is better than laziness II: Composing listless functions. In *Programs As Data Objects*, pages 282–305. Springer-Verlag, 1985. Lecture Notes in Computer Science 217.

[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.

[Wat] Richard C. Waters. To NReverse when consing a list or by pointer manipulation to avoid it; that is the question. *Lisp Pointers (to appear)*.

[Wat78] Richard C. Waters. Automatic analysis of the logical structure of programs. Technical Report AI-TR-492, MIT Artificial Intelligence Laboratory, December 1978.

[Wat79] Richard C. Waters. A method for analyzing loop programs. *IEEE Transactions on Software Engineering*, SE-5(3):237–247, May 1979.

[Wat84] Richard C. Waters. Expressional loops. In *ACM Symposium on the Principles of Programming Languages*, pages 1–10, 1984.

[Wat87] Richard C. Waters. Efficient interpretation of synchronizable series expressions. In *ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, volume SE-5, pages 74–85, 1987.

[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.

[Wat92] D. A. Watt. Modular description of programming languages. *The Computer Journal*, 35:A009–A0028, 1992.

[WCRS91] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, August 1991.

[WS73] S. A. Walker and H. R. Strong. Characterizations of flowchartable recursions. *Journal of Computer and System Sciences*, 7:404–447, 1973.

Appendix A

Glossary

This dissertation defines many new terms and uses some existing terms in a non-standard way. As an aid to the reader, this glossary contains definitions of these terms. Each entry includes a bracketed number that indicates a page on which the term is either introduced or discussed.

across [132] An orientation for a tile operation indicating that the operation is unordered with respect to subcall initiations and returns. Applies to both linear and tree-shaped tiles. Also used to describe the shape of a sliver exhibiting no inter-layer data dependencies.

aggregate data approach [82] An approach to the signal processing style of programming in which the signal processing operators are represented as procedures that manipulate aggregate data structures (e.g., lists, arrays, streams, trees).

arguments [131] The values consumed by a procedure application. In tile diagrams, they appear as inputs terminating on the line representing the call boundary.

attribute grammar [124] A grammar-based formalism for declaratively specifying the decoration of a tree with attributes.

barrion [192] A pair of a down synchron and an up synchron that represents a call boundary.

barrier synchronization [302] A style of synchronization in which all members of a group of processes must rendezvous at a point before any are allowed to proceed. Synchrons provide barrier synchronization for a dynamically determined number of processes.

between [148] An orientation for a tree tile operator indicating that the operator must be performed after some subcall return but before another subcall initiation.

between-LR [148] An orientation for a binary tile operator indicating that the operator must be performed after the left subcall return but before the right subcall initiation.

between-RL [148] An orientation for a binary tile operator indicating that the operator must be performed after the right subcall return but before the left subcall initiation.

binary down [152] Shape of a parallel binary tile or computation with at least one *up-both* operation.

binary tile [145] A tile in which at most two subcalls may be made.

binary trellis [145] An infinite binary tree structure representing the potential computation of a binary recursion. Each position of the trellis corresponds to the computation performed by one layer of the recursion.

binary up [152] Shape of a parallel binary tile or computation with *up-both* operations.

cable [65] In a sliver decomposition, a collection of wires that connects slivers.

call boundary [130] An abstract barrier that separates the computation of a procedure body from the computation of its arguments and the computation that uses its results. The call boundary is defined by two events: call initiation begins the computation of the body, and call return ends the computation of the body. In tile diagrams, a call boundary is represented as the top line of a tile.

call initiation [131] The event marking the start of a tile computation. All arguments must be available before this event.

call return [131] The event marking the end of a tile computation. All result values must be available before this events.

cell [298] A one-slot mutable data structure supported by OPERA.

channel approach [82] An approach to the signal processing style of programming in which the signal processing operators are represented as processes that transmit data over communication channels.

compaction approach [208] An approach to the filtering of aggregate data in which the output of the filter contains only the elements that pass the filter. Used in contrast with **gap approach**.

computational shapes [129] Computational patterns that characterize how processes unfold over time. See also **shape**.

consumpts [174] Horizontal inputs consumed by a subtile (in contrast with **arguments**, which are vertical inputs). In subtile diagrams, consumpts appear as inputs terminating on the left-hand edge of the subtile. In the context of slivers, consumpts refer to the input slags of a sliver.

conditionally tail-recursive [137] Describes a tile that contains both trivial and non-trivial paths through an *up* shard.

control arm [133] A section of a shard that contains a conditional that must be performed before some subcall.

deadlock [142] A state in which a computation can make no progress even though it has not terminated with a result.

down [132] An orientation for a linear tile operation indicating that the operation must be performed before subcall initiation. Also used to describe the shape of a sliver consisting of only down operations. *Down-both*, *down-left*, and *down-right* tree operations are loosely classified as *down* operations as well.

down barrier [166] A point defined by call initiation that separates the computation of the arguments to a procedure call from the computation of the body of the call.

down-both [148] An orientation for a binary tile operator indicating that the operator must be performed before both the left and right subcall initiations.

down-left [148] An orientation for a binary tile operator indicating that the operator must be performed before the left subcall initiation but is unordered with respect the right subcall.

down-right [148] An orientation for a binary tile operator indicating that the operator must be performed before the right subcall initiation but is unordered with respect the left subcall.

down **phase** [139] The phase of a linear iteration or recursion in which all *down* operations are performed.

DYNAMATOR [400] A graphical program animator for the EDGAR graph reduction model.

eagon [319] An OPERA object embodying an eager evaluation strategy. Equivalent to the **future** objects of other languages.

eager evaluation [158] A non-strict argument evaluation strategy for a procedure application in which arguments are evaluated in parallel with the evaluation of the procedure body.

EDGAR [347] An *E*xplicit *D*emand *G*r*A*ph *R*eduction model. EDGAR differs from most other graph reduction models in its explicit representation of the demand for the evaluation of a subgraph.

element wire [176] In a sliver decomposition, a wire that transmits the element associated with a particular layer of a recursive process.

excludon [314] A mutual exclusion object supported by OPERA.

fan-in [26] A situation in which a device has more than one input.

fan-out [26] A situation in which an output of some device is shared by more than one other device.

filton [336] A data structure representing a filtered element of a synchronized lazy list.

filtered slag [335] A synchronized lazy aggregate whose elements may be gaps.

future [298] Synonym of **eagon**.

gap [177] A distinguished element in an aggregate data structure that marks a position where a value has been filtered out. Written as **#g**.

gap approach [208] An approach to the filtering of aggregate data in which the output of the filter contains explicit gap entities for elements that did not pass the filter. Used in contrast with **compaction approach**.

in [152] Shape of a sequential tree tile having at least one *between* operation and no *up* operations. Also applies to tree slivers and computations whose operations satisfy these criteria.

in-order [269] A tree processing strategy in which the node of a tree is processed after some of its children but before other of its children.

interface tiles [143] Unreplicated tiles that perform initialization and finalization for a replicated tile computation.

intermediates [133] Values that do not cross the call boundary or subcall boundaries of a procedure. In tile diagrams, intermediates are produced by one shard and consumed by another shard within a tile.

iteration [51] Synonym for **linear iteration**.

layer decomposition [65] "Horizontal" decomposition of a recursive monolithic computation that collects together all operations performed in one layer of the recursion.

lazy data [98] Data structures whose components are not computed until their values are required.

lazy evaluation [158] A non-strict argument evaluation strategy for a procedure application in which the evaluation of arguments is suspended until their values are required.

lazon [315] An OPERA object embodying a lazy evaluation strategy. Equivalent to an automatically forced Scheme **delay** object.

left-to-right tile [150] Any tree tile in which dataflow constrains subcalls to be visited in a left-to-right order.

linear computation [51] A process whose dynamic calls form a linear sequence.

linear *down* shape [140] Shape of a linear computation that has no *up* phase. Iterative computations have a linear *down* shape.

linear iteration [51] A linear computation exhibiting constant control space.

linear recursion [51] A linear computation requiring an implicit control stack.

linear tile [130] Tile in which at most one subcall may be made.

linear *up* shape [140] Shape of a linear computation that has an *up* phase.

linear trellis [130] Infinite linear structure representing the potential computation of a linear iteration or recursion. Each position of the trellis corresponds to the computation performed by one layer of the iteration or recursion.

lock step component [171] A collection of slivers that engages in lock step processing to simulate the behavior of a monolithic recursive procedure. Any two slivers communicating via a synchronized lazy aggregate are in the same lock step component.

monolithic [52] Describes a program organization in which programming idioms are manually interwoven in a way that obscures the conceptual structure of the program.

modular [52] Describes a program organization that effectively encapsulates programming idioms into units that can be combined in mix-and-match ways.

multi-threaded [163] Describes a computation with a multiple loci of control.

non-strict evaluation [158] Any argument evaluation strategy in which the evaluation of arguments is not required to precede the evaluation of the body of the applied procedure. Laziness and eagerness are examples of non-strictness.

non-tail call [166] A procedure call that requires the pushing of control stack.

non-waiting pointer [302] A reference to a synchron that blocks a rendezvous from occuring at that synchron.

Opera [295] A dialect of Scheme supporting concurrency (concurrent evaluation of procedure call subexpresions), synchronization (synchrons and excludons), and non-strictness (lazons and eagons).

operational faithfulness [29] The principle that a lock step component of slivers should simulate the operational behavior generated by a monolithic recursive procedure.

operation order [29] A partial order describing the relative times at which operations may be performed.

orientation [132] A property of a tile operator that specifies when it is performed relative to the subcalls of the tile.

`par` [101] An eager evaluation mechanism introduced by Hughes. Similar to a **future**.

parallel down [164] Shape of a parallel binary tile or computation with no *up-both* operations.

parallel tile [150] Any tree tile in which there is no dataflow between subcalls.

parallel up [164] Shape of a parallel binary tile or computation with at least one *up-both* operation.

post [152] Shape of a sequential tree tile having at least one *up* operation. Also applies to a tree computation performing only these operations.

post-order [269] A tree processing strategy in which a node is processed after all of its children are processed.

pre [152] Shape of a sequential tree tile having only *down* operations. Also applies to a tree computation performing only these operations.

pre-order [269] A tree processing strategy in which a node is processed before any of its children are processed.

presence wire [176] In a sliver decomposition, a wire that transmits a flag indicating whether the current element is present. A presence wire carrying false indicates a gap.

products [174] Horizontal outputs produced by a subtile (in contrast with **results**, which are vertical outputs). In subtile diagrams, products appear as outputs originating on the right-hand edge of the subtile. In the context of slivers, refers to the output slags of a sliver.

recursion [450] Any recurisvely-structured computation requiring non-constant control space. See also **recursion**.

reify [212] In the context of filtering, an operator that turns a gap into a non-gap value.

rendezvous [197] The point in time at which all processes that *could* be waiting on a given synchron *are* waiting on that synchron.

results [131] The values produced by a procedure application. In tile diagrams, they appear as outputs originating at the line representing the call boundary.

reusability [29] The principle that a module (e.g., a sliver) should have a standard interface so that it can be reused in mix-and-match ways.

right-to-left tile [150] Any tree tile in which dataflow constrains subcalls to be visited in a right-to-left order.

sequential tile [150] Any tree tile in which dataflow constrains subcalls to be visited in some sequential order.

series [104] A synchronized sequence structure defined by Waters. It is the basis for a strategy that transforms modular sequence programs into efficient loops.

shape [138] A property that summarizes the operational nature of a computation. When applied to a tile or subtile, shape is a static property determined by the orientations of operators. When applied to the computation generated by a tile or subtile, shape is a dynamic property determined by the orienations of operators that are dynamically performed.

shard [132] The collection of operations within a tile or subtile that share the same orientation.

signal processing style [81] A style of organizing programs as signal processing block diagrams in which information flows through devices that generate, map, filter, and accumulate data. Aggregate data approaches and channel approaches are two common examples of the signal processing style. SPS is an abbreviation for this style.

single-threaded [163] Describes a computation with a single locus of control.

skeletal node [192] One of the nodes that forms the backbone of an aggregate data structure.

slag [190] Abbreviated form of *S*ynchronized *L*azy *AG*gregate.

sliver [65] Fragment of a recursive computation that captures a programming idiom. When used in the conjunction with synchronized lazy aggregates, denotes a procedure that manipulates such aggregates.

sliver decomposition [65] "Vertical" decomposition of a recursive monolithic computation that distributes the recursive structure over each of the components.

space profile [407] A function that describes the space required by a process as a function of execution time.

SPS [81] Acronym for *S*ignal *P*rocessing *S*tyle.

stream [99] An incrementally computed sequence. In this document, I use the term exclusively to mean Scheme's lazy lists.

strict data [409] Data structures that require their components to be fully computed values.

strict evaluation [131] Any argument evaluation strategy in which the evaluation of arguments is required to precede the evaluation of the body of the applied procedure.

subarguments [131] The values consumed by the subcall boundary of a tile.

subcall [130] Any procedure application that appears within the body of a given procedure.

subcall boundary [130] The call boundary associated with a subcall. In tile diagrams, a subcall boundary is represented as a line at the bottom of the tile. A single tile may have several subcall boundaries.

subcall initiation [131] The event that marks the beginning of a subcall's computation. All subarguments must be available before this time.

subcall return [131] The event that marks the end of a subcall's computation. All subresults are available after this time.

subresults [131] The values produced by the subcall boundary of a tile.

subtile [173] A tile fragment that specifies the computation performed by one layer of a sliver.

SYNAPSE [221] A language for manipulating synchronized lazy lists and trees. SYNAPSE is a suite of slag-manipulation procedures embedded in OPERA. The name stands for *SYN*chronized *A*ggregate *P*rogramming *S*tructures *E*nvironment.

`synch` [101] Hughes's construct for synchronizing two demands for a value.

synchron [302] An OPERA object permitting barrier synchronization of a dynamically determined number of processes. Synchrons are at the heart of the lock step processing model embodied by slivers and synchronized lazy aggregates.

synchronized lazy aggregate [190] A dynamically unfolding aggregate data structure carrying synchronization tokens that enable lock step processing among the producers and consumers of the aggregate.

synchronized lazy list [191] A synchronized lazy aggregate that denotes a sequence of elements.

synchronized lazy tree [191] A synchronized lazy aggregate that denotes a tree of elements.

syndrite [191] A synchronized lazy tree. The name stands for *SYN*chronized den*DRITE*.

synquence [191] A synchronized lazy list. The name stands for *SYN*chronized se*QUENCE*.

synter [192] A structure annotated with two synchrons that connects the skeletal nodes of a synchronized lazy aggregate. The name stands for *SYN*chronized poin*TER*.

tail call [169] A procedure call that does not require the pushing of control stack.

tail-recursive [131] Describes the tail call of a recursive procedure. When applied to tile operations or slivers, the term is synonymous with "*down* shaped".

termination wire [176] In a sliver decomposition, a wire that transmits a flag indicating whether there is any more data.

tile [130] A specification of the computation performed by one layer of an iteration or recursion. A tile is depicted as a box whose contents specify the potential operations performed by the layer.

tree shaped computation [56] A process whose dynamic calls form a tree.

unfiltered slag [330] A synchronized lazy aggregate whose elements may be gaps.

unitilable [138] Describes a recursive computation in which every layer is generated by the same tile.

unreify [212] In the context of filtering, an operator that turns a non-gap value into a gap.

up [132] An orientation for a linear tile operation indicating that the operation must be performed after subcall return. Also used to describe the shape of a linear sliver that has at least one up operation. *Up-both*, *up-left*, and *up-right* tree operations are loosely classified as *up* operations as well.

up barrier [166] A point defined by call return that separates the computation of the results of a procedure call from the computation that uses these results.

up **phase** [139] The phase of a linear recursion in which all *up* operations are performed.

up-both [148] An orientation for a binary tile operator indicating that the operator must be performed after both the left and right subcall returns.

up-left [148] An orientation for a binary tile operator indicating that the operator must be performed after the left subcall return but is unordered with respect the right subcall.

up-right [148] An orientation for a binary tile operator indicating that the operator must be performed after the right subcall return but is unordered with respect the left subcall.

waiting pointer [302] A reference to a synchron that indicates that a process is ready to engage in a rendezvous at that synchron.